

(19) 日本国特許庁 (J P)

(12) 公開特許公報 (A)

(11) 特許出願公開番号
特開2000-132408
(P2000-132408A)

(43) 公開日 平成12年5月12日 (2000.5.12)

(51) Int.Cl. ⁷	識別記号	F I	テーマコード* (参考)
G 0 6 F	9/455	C 0 6 F	3 1 0 A
	9/45		S
	11/34		3 2 2 F

審査請求 未請求 請求項の数12 O L (全 55 頁)

(21) 出願番号 特願平11-299576

(22) 出願日 平成11年10月21日 (1999. 10. 21)

(31) 優先権主張番号 0 9 / 1 7 6 1 1 2

(32) 優先日 平成10年10月21日 (1998. 10. 21)

(33) 優先権主張国 米国 (U S)

(71) 出願人 000003223

富士通株式会社

神奈川県川崎市中原区上小田中4丁目1番
1号

(72) 発明者 リチャード エー. レティン

アメリカ合衆国, ニューヨーク 10003,
ニューヨーク, フィフス アベニュー 25,
ナンバー6シー

(74) 代理人 10007/517

弁理士 石田 敬 (外4名)

最終頁に続く

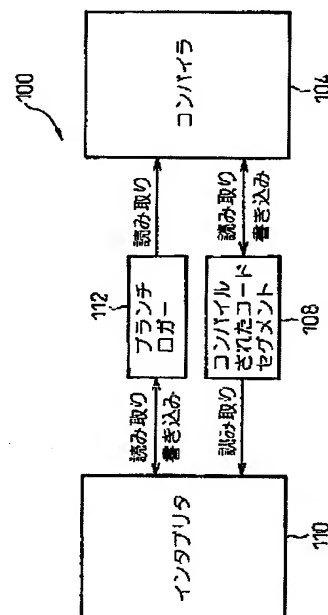
(54) 【発明の名称】 コンピュータアーキテクチャエミュレーションシステム

(57) 【要約】

【課題】 変換元オペレーティングシステム上で最適化
を実行しつつターゲットオブジェクトコードの動的コン
パイル及び変換を実行する。

【解決手段】 ターゲットコードのコンパイル及び最適
化は、実時間で動的に実施される。コンパイラは、32
ビットと言った大きいオーダーの命令を処理するホスト
プロセッサが、16ビット、18ビットと言った、より
小さいオーダーの命令を処理するターゲットプロセッサ
をエミュレートできるように、テンプレートベースの変
換及び解釈に関するエミュレーションを改善する分析及
び最適化を行う。プログラム実行の間、変換プログラム
はブランチ動作を記録する。ブランチ動作が実行された
回数が閾値を越えたとき、ブランチの変換先はコンパ
イルのためのシードとなり、シードとシードの間のコード
部分はセグメントとして定義される。

図 1



【特許請求の範囲】

【請求項1】 変換先コンピュータアーキテクチャシステム上で変換元コンピュータアーキテクチャをエミュレートするコンピュータアーキテクチャエミュレーションシステムであって、

変換元オブジェクトコードを対応する変換されたオブジェクトコードにそれぞれ変換し、該変換元オブジェクトコードのブランチ命令の実行数を決定するインタプリタと、

対応するブランチ命令の実行数が閾値を越えたときに該変換元オブジェクトコードの命令をセグメントにグループ化し、該セグメントを動的にコンパイルするコンパイラと、

を具備するコンピュータアーキテクチャエミュレーションシステム。

【請求項2】 前記インタプリタ及び前記コンパイラは、実時間でマルチタスキングオペレーティングシステムにて同時に動作するタスクである、請求項1に記載のコンピュータアーキテクチャエミュレーションシステム。

【請求項3】 変換先コンピュータアーキテクチャシステム上で変換元コンピュータアーキテクチャをエミュレートするコンピュータアーキテクチャエミュレーションシステムであって、

変換元オブジェクトコードを、対応する変換されたオブジェクトコードに、それぞれ変換すると共に、それぞれが変換されたオブジェクトコード命令の実行の間に、実時間で変換元オブジェクトコードのブランチ情報をプロファイルする複数のインタプリタと、

前記複数のインタプリタの何れかからの変換元オブジェクトコード命令を、変換元オブジェクトコードに於ける対応するブランチ命令に基づいてセグメントにグループ化し、対応するブランチ命令の実行数が閾値より大きいとき、変換元オブジェクトコードのセグメントを動的にコンパイルするコンパイラと、

を具備するコンピュータアーキテクチャエミュレーションシステム。

【請求項4】 前記複数のインタプリタの各々は、ブランチ命令をプロファイルすると共に、閾値を越えなかったブランチ命令を、ブランチログをコールして記憶する、請求項3に記載のコンピュータアーキテクチャエミュレーションシステム。

【請求項5】 変換先コンピュータアーキテクチャシステム上で変換元コンピュータアーキテクチャをエミュレートするコンピュータアーキテクチャエミュレーションシステムであって、

変換元オブジェクトコードを対応する変換されたオブジェクトコードにそれぞれ変換するインタプリタであって、変換元オブジェクトコードのブランチ命令を、各ブランチ命令に関する実行数を記憶すると共にその実行数

を閾値と比較することによって、プロファイルし、閾値を越えたブランチ命令をシードとして指定するインタプリタと、

該シードに基づいて、変換元オブジェクトコード命令をセグメントにグループ化し、前記インタプリタによる変換及びプロファイリングの間に、変換元オブジェクトコードのセグメントを動的にコンパイルするコンパイラと、

を具備するコンピュータアーキテクチャエミュレーションシステム。

【請求項6】 各セグメントは、対応するシードに基づいて変換元オブジェクトコードを最適化した結果得られた命令を含み、

各セグメントは、単位として導入及び非導入される、請求項5に記載のコンピュータアーキテクチャエミュレーションシステム。

【請求項7】 前記インタプリタによって決定されたブランチ命令のブランチプロファイル情報を記憶するブランチログを更に具備し、該ブランチプロファイル情報は、ブランチアドレス、ブランチサクセサ、非ブランチサクセサ、ブランチ実行カウント、及びブランチテイクンカウントを含むとともに、ブランチ命令のエミュレーションの間に、前記インタプリタによって記録される、請求項6に記載のコンピュータアーキテクチャエミュレーションシステム。

【請求項8】 命令実行処理が所定の実行率でタイムリーに実行されていない場合に、前記コンパイラは、プロファイルを用いて実行状態を追跡し、ブランチカウントが所定数を下回っているか否かをチェックし、ページフォールトのようなブランチ情報を記録するためのオブジェクトコードを作成する、請求項6に記載のコンピュータアーキテクチャエミュレーションシステム。

【請求項9】 実行数を含む変換元オブジェクトコードにおけるブランチ命令のプロファイル情報を記憶するブランチログであって、頻繁に実行されるブランチ命令のプロファイル情報を記憶するキャッシュと、頻繁には実行されないブランチ命令のプロファイル命令を記憶するブランチログと、を含むブランチログ、を更に具備し、

プロファイル情報は、ブランチアドレス情報とブランチ変換先情報とを組み合わせるキャッシュに組織されるとともに、該プロファイル情報は、複数のグループに、該グループへのエントリの降順に記憶される、

請求項6に記載のコンピュータアーキテクチャエミュレーションシステム。

【請求項10】 前記コンパイラは、変換元オブジェクトコードのセグメントを選択し、該シード及び該ブランチのプロファイル情報に基づいてコンパイルするブロックピッカであって、オリジナル命令を記述する制御フローグラフを生成し、そのグラフをコン

パイルするブロックピッカと、
該制御フローグラフを命令の線形リストへと平坦化する
ブロックレイアウトユニットと、
オリジナル命令を変換されたコードセグメント命令に実
際にコンパイルする最適化コード発生ユニットと、
を更に含む、請求項6に記載のコンピュータアーキテク
チャエミュレーションシステム。

【請求項11】 多重タスキング変換先コンピュータアー
キテクチャ上で変換元コンピュータアーキテクチャを
エミュレートする多重タスキングコンピュータアーキテ
クチャエミュレーションシステムであって、
変換元オブジェクトコードを対応する変換されたオブジ
ェクトコードにそれぞれ変換し、変換元オブジェクトコ
ードのブランチ命令の実行数を決定するインタプリタタ
スクと、
多重タスキング変換先コンピュータアーキテクチャ上で
前記インタプリタタスクと共に動作するコンパイラタス
クであって、対応するブランチ命令の実行数が閾値を越
えたとき変換元オブジェクトコードの命令をセグメント
にグループ化し、このセグメントを動的にコンパイルす
るコンパイラタスクと、
を具備する多重タスキングコンピュータアーキテクチャ
エミュレーションシステム。

【請求項12】 前記多重タスキングコンピュータアー
キテクチャエミュレーションシステムは動的変換システ
ムであり、前記多重タスキングコンピュータアーキテク
チャシステムは、
前記インタプリタタスクによって送られるコンパイル要
求の率と、前記コンパイラタスクによって完成されるコ
ンパイルの率とを、閾値を変えることによってコンパイ
ラタスクが遊休状態に入らないようにしつつ、等しくす
るソフトウェアフィードバック、
を更に具備する、請求項11に記載の多重タスキングコ
ンピュータアーキテクチャエミュレーションシステム。

【発明の詳細な説明】

【0001】

【発明の属する技術分野】本発明は、第2のオペレーテ
ィングシステムをエミュレートするため、ホスト処理シ
ステム上で動作するオブジェクトコード変換プログラムの
技術に関し、更に詳しくは、ホストプロセッサオブジ
ェクトコード命令セットを有するホストプロセッサの実
行時に、オリジナルオブジェクトコード命令セットの分
析及び計算を実時間で行う動的オブジェクトコード変換
プログラムの技術に関する。

【0002】

【従来の技術】オブジェクトコード変換プログラムの分
野では、1つのコンピュータ用に開発されたオブジェク
トコードを、異なるコンピュータアーキテクチャを有す
る別のコンピュータ用に変換することが必要になる。そ
うしたオブジェクトコード変換方法には、“静的オブジ

ェクトコード変換方法”と称する従来の方法があり、こ
の方法では、命令文はその実行以前に、先ず第2のコン
ピュータアーキテクチャのオブジェクトコードに変換さ
れる。また、第2の従来方法としては、“動的オブジェ
クトコード変換方法”と称する方法があり、この方法で
は、命令を実行する間に、第1のオブジェクトコードを
第2のオブジェクトコードに変換する。

【0003】静的オブジェクトコード変換方法の技術で
は、実行時間は変換に要する時間による影響を受けな
い。しかし、静的オブジェクトコード変換の実行に当た
っては、変換されたオブジェクトコードの物理サイズは
大きくなる。言い換えれば、静的オブジェクトコード変
換方法では、変換されたオブジェクトコードの操作ステ
ップ数の増加は避けられない。その結果、変換されたオ
ブジェクトコードのパフォーマンスは悪くなり、非能率
となる問題がある。

【0004】他方、動的オブジェクトコード変換方法で
は、静的オブジェクトコード変換方法に較べて、変換さ
れたオブジェクトコードの物理サイズは相対的に小さく
なる。しかし、この従来型の動的オブジェクトコード変
換方法は、まれにしか使用しないオブジェクトコードも
含めて、全てのオブジェクトコードが変換されてしまう
と言う問題を抱えている。言い換えれば、従来型動的オ
ブジェクトコード変換方法は、何回も実行されるオブジ
ェクトコードの効率的な認識ができず、効率を犠牲にし
て、オリジナルオブジェクトコードの変換に必要な時間
を増加せしめている。

【0005】

【発明が解決しようとする課題】したがって、本発明の
目的は、従来技術の諸問題に取り組むと共に、変換され
たオブジェクトコードの動的最適化を行うオブジェクト
コード変換プログラムを提供することにある。

【0006】また、本発明の他の目的は、コンパイラが
コンパイルを完了するまで、主要プログラムをプロファ
イルすることである。コンパイラはこのプロファイルを
用いて、主要プログラムをコンパイルし、かつ最適化す
る。

【0007】また、本発明の更に他の目的は、動的最適
化及びコンパイル時に、変換されていないコードから変換
されたコードへのジャンプ動作を可能にすることであ
る。

【0008】また、本発明の更に他の目的は、コンパイ
ラに送られた変換要求数と変換完了数との差を計算する
ソフトウェアフィードバックを備えた動的最適化オブジ
ェクトコード変換プログラムを提供することである。

【0009】また、本発明の更に他の目的は、1つのマ
シン語表現によるコンピュータプログラムをその実行時
に、他のマシン語表現によるコンピュータプログラムに
動的に変換することである。

【0010】更にまた、本発明の目的は、変換元オブジ

ェクトコードのブランチ（分岐）に対応する複数のシードから変換用のセグメントを決定する動的オブジェクトコード変換プログラムを提供することである。

【0011】

【課題を解決するための手段】本発明の目的は、変換先コンピュータアーキテクチャシステムに関して、変換元コンピュータアーキテクチャをエミュレートするコンピュータアーキテクチャエミュレーションシステムによって達成される。このシステムは、変換元オブジェクトコードを対応する変換されたオブジェクトコードにそれぞれ変換し、変換元オブジェクトコードのブランチ命令の実行数を決定するインタプリタと、対応するブランチ命令の実行数が閾値を越えたとき、変換元オブジェクトコードの命令をセグメントにグループ化し、このセグメントを動的にコンパイルするコンパイラとを含んでいる。

【0012】更に、本発明の目的は、変換先コンピュータアーキテクチャシステムに関して、変換元コンピュータアーキテクチャをエミュレートするコンピュータアーキテクチャエミュレーションシステムによって達成される。このシステムは、変換元オブジェクトコードを、対応する変換されたオブジェクトコードに、それぞれ変換すると共に、それぞれが変換されたオブジェクトコード命令の実行の間に、実時間で変換元オブジェクトコードのブランチ情報をプロファイルする複数のインタプリタと、これら複数のインタプリタの何れかからの変換元オブジェクトコード命令を、変換元オブジェクトコードに於ける対応するブランチ命令に基づいてセグメントにグループ化すると共に、対応するブランチ命令の実行数が閾値より大きいとき、変換元オブジェクトコードのセグメントを動的にコンパイルするコンパイラと、を含んでいる。

【0013】また更に、本発明の目的は、変換先コンピュータアーキテクチャシステムに関して、変換元コンピュータアーキテクチャをエミュレートするコンピュータアーキテクチャエミュレーションシステムによって達成される。このシステムは、変換元オブジェクトコードを対応する変換されたオブジェクトコードにそれぞれ変換すると共に、変換元オブジェクトコードのブランチ命令を、各ブランチ命令に関する実行数を記憶し、その実行数を閾値と比較することによってプロファイルし、閾値を越えたブランチ命令をシードとするインタプリタと、このシードに基づいて、変換元オブジェクトコードの命令をセグメントにグループ化し、上記インタプリタによる変換及びプロファイリングの間に、変換元オブジェクトコードのセグメントを動的にコンパイルするコンパイラと、を含んでいる。

【0014】また更に、本発明の目的は、多重タスキング変換先コンピュータアーキテクチャに関して、変換元コンピュータアーキテクチャをエミュレートする多重タスキングコンピュータアーキテクチャエミュレーション

システムによって達成される。このシステムは、変換元オブジェクトコードを対応する変換されたオブジェクトコードにそれぞれ変換し、変換元オブジェクトコードのブランチ命令の実行数を決定するインタプリタと、対応するブランチ命令の実行数が閾値を越えたとき、変換元オブジェクトコードの命令をセグメントにグループ化し、このセグメントを動的にコンパイルする多重タスキング転送先コンピュータアーキテクチャの上記インタプリタと共に動作するコンパイラと、を含んでいる。

【0015】本発明のこれら及びその他の目的、並びにその利点は、添付図面を参照して以下に述べる本発明の好適実施形態の説明から容易に理解されよう。

【0016】

【発明の実施の形態】好適実施形態の詳細な説明
添付図面にその例を示す本発明の好適実施形態について詳細に説明する。尚、添付図面中、同一参照番号は同一要素を示すものとする。

【0017】本発明の第1実施形態

1. システムの概観

本発明は全体として最適化オブジェクトコード変換プログラム(optimizing object code translator)（以下、“OOC”と言う）に関し、プログラムはコンピュータアーキテクチャエミュレーションシステムの一部として、マイクロプロセッサ命令セットの動的コンパイルを実行する。実行時以前には、アプリケーション命令セットへの単純なアクセスもないから、コンパイルは動的である。オブジェクトコード変換システムの一部としてコンパイラを使用することによって、このシステムはテンプレートベースの変換及びテンプレートベースの解釈(interpretation)に関するエミュレーション性能を改善する分析及び最適化を行うことができる。

【0018】エミュレーション用のホストプロセッサは、例えばインテル社のPentium Pro等、市場で入手可能なものが好ましい。Pentium Pro の命令セットアーキテクチャは、サイズの異なるデータを容易に操作できるので、16ビット及び18ビットのオブジェクトコード命令両者のエミュレーションが楽に行える。16ビット及び18ビットオブジェクトコード命令は、第2のプロセッサ、例えば富士通社のK-シリーズプロセッサ側のオリジナルアプリケーションに対して設計することもできる。

【0019】意味のあるコンパイラ型最適化の実施は、命令のフローグラフに関する情報が有る時に、はじめて可能になる。伝統的なコンパイラでは、最適化を開始する以前に、全ルーチンが完全に解析されるから、フローグラフが与えられ、そして十分に定義される。OOCの場合にはこれとは違って、プログラムを走らせる前には、メモリイメージに於ける命令の記憶場所は未知である。このことは、命令はその長さを種々変化すると共に、非命

令データセットを任意に介在させているからである。全ての結合点が命令中にあるため、命令の記憶場所は未知となる。

【0020】それ故、フローグラフを決めるには、プログラムを走らさなければならない。初めに、インタプリタがプログラムを走らせる。インタプリタはプログラムの実行に当たって、1つのブランチオペレーションを実行する毎に、それに関する情報をOOCに報告する。この情報ロギング（記録）から、幾つかの命令及び幾つかの結合点が識別される。プログラムの進行につれて、フローグラフに関する情報は全体として決して完全とは言えないまでも、より完全になって行く。OOCシステムは、フローグラフについての部分情報によっても動作するように設計されている。即ち、システムは最適化を潜在性のある不完全なフローグラフで実行し、より多くの情報が利用可能になったとき、最適化コードをそれと交換できるように構成されている。

【0021】動的コンパイルは、インタプリタによって収集されたプロファイリング情報に基づいて、テキストのどの部分を最適化すべきかを選択する。或るブランチの実行回数が閾値を越えた時、そのブランチの変換先がコンパイル用のシード(seed)となる。このシードは、1つのユニットとしてコンパイルされるK命令の一部分を解析するための始点である。このユニットをセグメントと呼ぶ。

【0022】セグメントは、シードからオリジナルプロセッサ命令を最適化した結果生じるホストプロセッサ命令を含んでいる。セグメントは1つのユニットとして導入されたり、導入されなかったりする。インタプリタがOOCをコールして、ブランチについて報告すると、OOCはその報告に変換先コードが在れば、制御をセグメントに移行することを選択する。同様に、セグメントは制御をインタプリタへ返却移行するためのコードを含んでいる。

【0023】セグメント自身は、オリジナルプログラムからの可能なフロー経路のサブセットを表すだけの不完全なものであっても良い。しかし、この不完全な代表性は、エミュレーションの正確な動作を妨げるものではない。オリジナルコードを介して新たな予想外のフロー経路が生じた場合には、制御フローはインタプリタに戻り、その後、同セグメントは新たな制御フローのために置き換えられる。

【0024】II. OOCコードの構造

本発明の実施形態によれば、OOCは従来のオペレーティングシステム環境、例えばウインドウズ環境下で動作する。しかし、本発明の第2実施形態によれば、第2のオペレーティングシステム、例えば富士通社のKOI オペレーティングシステムのエミュレーションファームウェアとリンクして構成することもできる。

【0025】III. アーキテクチャ

図1は、OOCシステム100の高レベルアーキテクチャを示す。また、図1は、2つのタスク、即ちインタプリタ110及びコンパイラ104を示す。多重タスクオペレーティングシステムの下では、インタプリタ110及びコンパイラ104は同時に動作する。2つのタスクは共にブランチローガー112によってブランチログにアクセスすることができ、またコンパイルされたコードセグメント108にもアクセスすることができる。更に、インタプリタ110はコンパイラに対してコンパイル要求を送ることができる。2つのタスク間に於ける通信に関しては、以下の通信に関する項でより完全な説明を行う。

【0026】コンパイル制御フロー

図2は、オリジナルコードの1つのセクションをコンパイルするための制御フローと共に、OOC100の主要構成要素を示す。主なOOC処理段階は以下の通りである。先ず、インタプリタ110はブランチローガー112と交信して、ブランチ情報をプロファイルする。次いで、ブランチローガー112はシード選択法を用いて、どのシードをコンパイラ104に送るかを決定する。次に、ブロックピッカ114はシードとブランチプロファイル情報を用いて、コンパイルするオリジナルコードの1つのセクションを選択する。次いで、ブロックピッカ114は、コンパイルするオリジナル命令を描く制御フローグラフ(CFG)を生成し、このCFGをブロックレイアウトユニット116に渡す。

【0027】次いで、ブロックレイアウトユニット116は、制御フローグラフを命令の線形リストに単層化する。最適化コード発生ユニット118は、オリジナル命令を実際にコンパイルし、変換コードセグメント命令を生成する。この生成された変換コードは最終的には、変換されるセグメントに関する情報と共に、セグメント導入ユニット120に渡される。このユニット120は、このコードをインタプリタ110が利用できるようにする。

【0028】OOC実行時の制御フロー

図3は、OOCの通常実行時に於ける制御フローを示す。インタプリタ110がコードを実行している間に、或る命令を実行する際には、OOCはブランチローガー112に入ることができる。ブランチローガー112は、インタプリタ110に戻るか、或いはブランチの変換先が既にコンパイルされている場合には、コンパイルされたコードの導入されたセグメントの1つに入ることができる。このコンパイルされたコードから、セグメントからセグメントへの遷移をするか、又はインタプリタ110へ戻ることができる。コンパイルされたコードは、インタプリタ110をコール（呼出）して単一オリジナル命令を実行するか、又はインタプリタ110にジャンプして全ての制御をインタプリタ110に渡すことができる。

【0029】本願の第1実施形態に関する説明は、以下

の各項に分けて行う。第1の項では、インタプリタ110とコンパイラ104のインタフェースについて述べる。第2の項では、OOC T用のインタプリタ110について実施した修正について述べる。第3の項では、コンパイラ104について述べる。そして最後の項では、ウィンドウズのテスト環境について説明する。

【0030】本発明の第2実施形態から第9実施形態までの説明は、第1実施形態の説明の後に続けて行う。

【0031】Ⅳ. 通 信 (共通ユニット)
インタプリタ110とコンパイラ104は、幾つかの方法で相互に通信を行う。インタプリタ110は、ブランチローガ112と交信することによって、ブランチ情報をブランチログに記録する。また、コンパイラ104はブランチログを読むことが可能である。コンパイラ104はコンパイルされたコードセグメントを生成し、それらを変換テーブル中のそれぞれのエントリポイントに格納する。インタプリタ110は変換テーブルを読むことができる。また、インタプリタ110はコンパイラ104に対し、バッファを介してシードアドレスを送る。この交信にコンパイラ104及びインタプリタ110両者

が用いる変換元コードは、共通ディレクトリにある。この項では、如何に通信が行われるかについて述べる。

【0032】共用OOC Tバッファ

コンパイラ104とインタプリタ110との間の全ての通信は、大きな共用メモリ領域であるOOC Tバッファを介して行われる。また、或る通信では、インタプリタ110とコンパイラ104の間でメッセージのやりとりをするのに、システムコールを用いる。

【0033】以下に述べるテーブル1は、OOC Tバッファの静的割当 (配分) 部分を示す。バッファの残りの部分は、異なるデータ構造用として以下のテーブル2に示すように動的に割当られる。OOC Tバッファの静的割当部分の中の或るフィールドは、動的割当部分のデータ構造を指す。これらのポインタは、それらが何を指しているかを示す上付数字を有している。例えば、静的割当部分のゾーンフィールドは数字2を有し、そしてゾーンフィールドは、数字2を有する動的割当部分のゾーンメモリデータ構造を指す。

【0034】

【表1】

テーブル1 : OOC Tバッファの静的割当部分

フィールド	オフセット	内 容
jump table	0h	インタプリタ110のエントリポイントアレイ、例えばIC_FETCH02、IU_PGMxx等。OOC T_INIT はこれらを書き込み、コンパイラ104はこれらを読み取る。コンパイラ104はインタプリタ110へのジャンプを発生するのにこれらを用いる。
trans_master_target_table ^T	1000h	ポインタアレイ、ASPアドレスの各ページについて1つ与えられる。ASPが使用しないページのポインタは0。ASPが使用するページのポインタはOOC Tバッファの動的割当部分のアレイを指す (以下参照)。
unallocated	41004h	バッファの動的割当部分の第1 不使用バイトを指すポインタ。初期化時にのみ使用。
length_left	41008h	バッファの動的割当部分の残りのバイト数。初期化時にのみ使用。
num_execs	4100Ch	インタプリタ110の数。

【0035】

【表2】

テーブル1 (続き)

フィールド	オフセット	内 容
zones ²	41010h	OOCTバッファの動的割当部分に於けるゾーンメモリに対するポインタ。OOCT_INITはこれを書き込み、コンパイラ104はこれを読み取る。コンパイラ104はコンパイル動作時にゾーンメモリを使用する。
zone_length	41014h	ゾーンメモリ量。OOCT_INITによって書き込まれ、コンパイラによって読み取られる。
segments ³	41018h	OOCTバッファの動的割当部分に於けるセグメントメモリに対するポインタ。OOCT_INITはこれを書き込み、コンパイラ104はこれを読み取る。コンパイラ104はセグメントメモリを用いてコンパイルされたコードを記憶する。
segments_length	4101Ch	セグメントメモリ量。OOCT_INITによって書き込まれ、コンパイラによって読み取られる。
branch_ll_tables ⁴	41020h	OOCTバッファの動的割当部分に於けるレベル1 (L1) ブランチキャッシュ構造に対するポインタ。
branch_record_free_list ⁵	41024h	OOCTバッファの動的割当部分に於ける不使用のBRANCH_RECORD 構造リスト。

【0036】

【表3】

テーブル1 (続き)

フィールド	オフセット	内 容
branch_header_table ⁶	41028h	BRANCH_RECORD 構造を含むハッシュテーブル。このテーブルはOOCTバッファに動的に割当てられる。
branch_log_lock	4102Ch	ブランチログに書き込むために保持されねばならないロック。
branch_seed_buffer	41030h	インタプリタ110がコンパイラ104にシードを送るのに使用するバッファ。
num_monitor_seed_messages	41060h	インタプリタ110がコンパイラ104に送ったメッセージ数を告げるカウンタ。但し、この段階でコンパイラ104の動作は完了していない。
seed_threshold_mode	41064h	インタプリタ110にシードのピックアップ方法を示すフラグ。シードはOOCT_DEBUG_MODE 又はOOCT_PERFORMANCE_MODE の何れかである。
seed_production_threshold	41068h	ブランチの宛先がコンパイラ104のシードとなる以前に、ブランチが実行しなければならない回数の閾値。
trickle_flush_ll_rate	4106Ch	ブランチがキャッシュからフラッシュ(クリア)され、メモリに書き戻される以前に、ブランチをL1キャッシュに於いて更新できる回数。
seeds_sent	41070h	不使用
seeds_handles	41074h	不使用

【0037】

【表4】

テーブル1 (続き)

フィールド	オフセット	内 容
exit	41078h	コンパイラ104はこのフラグを用いて、インタプリタ110に対し、コンパイラ104が信号受信後遮断したことを告げる。
segment_exit	4107Ch	コンパイルされたコードがexitにジャンプするインタプリタ110のエントリポイント。このエントリポイントに於けるコードは必要に応じてロックを解く。
segment_exit_interp	41080h	コンパイルされたコードが解釈せねばならない命令と共に終了点にジャンプするインタプリタ110のエントリポイント。このエントリポイントに於けるコードは必要に応じてロックを解除する。
segment_exit_log	41084h	コンパイルされたコードが非固定ブランチ命令と共に終了点にジャンプするインタプリタ110のエントリポイント。このエントリポイントに於けるコードは必要に応じてロックを解除する。
sbe_impl	41088h	コンパイルされたコードがSBE命令を実行するためにコールするインタプリタ110のエントリポイント。
cc_impl	4108Ch	コンパイルされたコードがCC命令を実行するためにコールするインタプリタ110のエントリポイント。
mv_impl	41090h	コンパイルされたコードがMV命令を実行するためにコールするインタプリタ110のエントリポイント。

【0038】

【表5】

テーブル1 (続き)

フィールド	オフセット	内 容
mv_impl_same_size	41094h	両ストリングの長さが同じ時、コンパイルされたコードがMV命令を実行するためにコールするインタプリタ110のエントリポイント。
segment_lock_mousetrap	41098h	コンパイルされたコードが依然としてロックを保持しているかを確認するためにコールするインタプリタ110のエントリポイント。デバッグ時にのみ使用。
breakpoint_trap	4109Ch	コンパイルされたコードがデバッガを停止するためにコールするインタプリタ110のエントリポイント。デバッグ時にのみ使用。
segment_gates	410A0h	SEGMENT_GATE 構造の配列。SEGMENT_GATE はコンパイルされたコードのセグメントをロックするのに使用する。
gate_free_list	710A0h	現在未使用のSEGMENT_GATE リスト。
ooct_stack_bottom ⁷	710A4h	コンパイラ104のスタックの最下位アドレス。OOCTバッファの動的割当部分内のポイント。
ooct_stack_top ⁷	710A8A	コンパイラ104のスタックの最上位アドレス。OOCTバッファの動的割当部分内のポイント。

【0039】

【表6】

テーブル1 (続き)

フィールド	オフセット	内 容
build_options	710ACh	インタプリタ110を構成するのに用いられるオプション。OOCT_compiler_start では、コンパイラ104はインタプリタが同じオプションで構成されたかをチェックする。
code_zone ²	710B0h	動的に割当られたメモリ領域に対するポインタ。コンパイラ104はこのメモリを用いてターゲット命令アレイを一時的に生成する。コンパイル終了時、このアレイはセグメントメモリにコピー後、削除される。

【0040】OOCTバッファの動的割当部分に於いては、データ構造のサイズは幾つかの変数に依存する。その1つは、オリジナルプロセッサのためのオペレーティングシステム、例えば富士通社のASPが使用するシステムページの数である。変換対象となる命令を含むASPアドレススペースの各ページに対して、変換テーブルには1つの変換されたページがある。もう1つの変数は、システムが記録(ログ)することを予想するブランチ命令の数である。現在、BRANCH_RECORDアレイ及びブランチヘッダテーブルのサイズに影響を与えるブランチ命令数

は、 2^{20} と予想する。各タスク毎に1つのキャッシュがあるから、インタプリタ110の数は、L1ブランチローガーキャッシュのサイズに影響を与える。

【0041】図4は、各種変数を設定した時のOOCTバッファの概略図である。この図4に於いて、ASPのページ数はASP命令について10MB、インタプリタ110の数は4、そしてOOCTバッファ全体のサイズは128MBである。

【0042】

【表7】

テーブル2: OOCTバッファの動的割当部分

名 称	内 容
Translation Table ¹	ASPによって使用される各ページに対して、変換テーブルに於いて1ページ当たり16KBが割当てられる。 サイズ=システムページ数*16KB。
BRANCH_RECORD array ²	ASPに於いて幾つのブランチ命令が発生するかを予想する(現在 2^{20} を予想)、そして各々について1つのBRANCH_RECORDを割り当てる。 サイズ= $2^{20} * 24$ バイト=24MB。
Branch header table ³	各予想ブランチに関するBRANCH_RECORDに対して1つのポインタがある。 サイズ= $2^{20} * 4$ バイト=4MB。
Branch L1 caches ⁴	各インタプリタ110には、4つのBRANCH_L1_RECORDから成るセットを32セット有する1つのキャッシュがある。 サイズ=Num execs * 32 * 4 * 24バイト。 最大サイズ= $16 * 32 * 4 * 24$ バイト=49152バイト。
OOCT stack ⁷	1MBスタック。
Zone memory ²	残余メモリのパーセンテージはゾーンメモリ用に使用される。現在、メモリの50%が使用される。
Segment memory ⁸	残余メモリのパーセンテージはセグメントメモリ用に使用される。現在、メモリの50%が使用される。

【0043】ブランチログ(ブランチローガー112)ブランチログデータ構造は、BRANCH_RECORDアレイ、ブランチヘッダテーブル、及びブランチL1キャッシュから成っている。ブランチローガー112の動作説明については、以下に述べるインタプリタ修正に関する項を参照されたい。この項では、インタプリタ110からコンパイラ104への情報通信に関して、ブランチログがどのように使われるかを説明する。

【0044】図4は、初期化の後のOOCTバッファを示す。種々の領域のサイズは基準化して示してある。例えば、OOCTバッファのサイズは128MB、ASPページ数は2560、インタプリタ110の数は2、及びブランチ命令の予想数は220として示してある。

【0045】コンパイラ104は、ブランチログから、条件付きブランチ命令が何回実行され、また何回実行されなかったかを読み取る。コンパイラは、この情報を2

つの方法で使用する。第1に、コンパイラ104が命令を解析するとき、コンパイラは、最も頻繁に実行された命令だけの解析をする。条件付きブランチ命令が発生した場合、コンパイラは、それが何回分岐したか、何回分岐しなかったかをチェックする。第2に、コンパイラ104がコードを発生するとき、コンパイラはブランチ命令の直ぐ後に条件付きブランチの最も相応しいサクセサ（後継）命令を置く。これによって発生コードの高速実行を可能にする。どのサクセサが最も相応しいかを決めるのに、コンパイラ104はブランチログ情報を利用する。詳細については、以下に述べるコンパイラ情報を参照されたい。

【0046】BRANCH Get _ Record(ooct/compiler/branch.c)

コンパイラ104がブランチログ情報を読み取る際、コンパイラはブランチ命令のアドレスを用いてBRANCH_Get _ Record手続きをコールする。この手続きは、ブランチログ内のブランチを参照し、BRANCH_RECORDアレイの要素の1つにポインタを戻す。斯うして、コンパイラ104は、ブランチ命令が何回実行されたか、何回分岐されたか、そして実行も、分岐もされなかったか回数を知ることができる。

【0047】変換テーブル (TRANS UNIT)

変換テーブルは、ASPアドレススペースの全ての命令に関する情報を含んでいる。変換テーブルは、命令がブランチの変換先であるか否か (JOIN)、命令がコンパイラ104にシードとして送られたか否か (BUFFERED)、及びセグメントに対するコンパイルされたコードのエントリポイントがあるか否かを記録する (ENTRY)。OOCTが初期化されると、変換テーブルはクリアされる。ブランチ情報が記録されると、それらの変換先がJOINポイントとして記録される。もしブランチが閾値より多くの回数実行すると、変換先はコンパイラ104にシードとして送られ、変換テーブルエントリはBUFFEREDとマークされる。コンパイラ104が変換されたバージョンをコンパイルし終わった後、コンパイラは、変換テーブルのエントリポイントのアドレスを記憶し、それらをENTRYとして記録する。

【0048】図5 (a)、(b) 及び (c) は、本発明の好適実施形態による変換テーブルの構造を示す。図5 (a) に示すように、ASPアドレスは2つの部分に分かれている。高位20ビットはページ番号、そして低位12ビットはページオフセットである。

【0049】図5 (b) は、ページ番号が第1レベル変換テーブルに於いてインデックスとして利用されることを示している。ASPが作用するページは第1レベルページである。ASPが使用しないページは、そのページ番号を持つ命令が決してないから、ポインタを有することはない。ポインタは第2レベルの変換テーブルを指す。ページオフセットをポインタに加えることによ

て、変換テーブルのエントリが与えられる。

【0050】図5 (c) が示すように、各エントリは32ビットの長さを持ち、そのフィールドは図の底部に示されている。第1ビットは、ASP命令がジョインポイントか否かを示す。第2ビットは、命令のためのセグメントエントリポイントがあるか否かを示す。第3ビットは、命令がコンパイラ104に対してシードとして送られたか否かを示す。変換テーブルエントリの他のビットは、もし1で有れば命令に関するエントリポイントのアドレスであり、エントリポイントがなければゼロである。

【0051】Kマシンアーキテクチャは可変長命令を有しているから、変換テーブルは命令中央にあるアドレス及びデータアドレスを含む全てのASPアドレスに対するエントリを有している。このことはテーブルを非常に大きくするが、アドレスに対して変換テーブルエントリを位置づける仕事を簡単にする。変換テーブルの構造は図5 (a)、(b) 及び (c) に示されている。上記のように、第2レベルの変換テーブルは、全てのASPアドレスに対して32ビットのエントリを有している。それ故、ASPが10MBのスペースを使用する場合には、第2レベル変換テーブルは40MBを使用する。変換テーブルのエントリを読み取り、書き込みする手続き及びマクロは幾つかある。即ち、

【0052】TRANS Set _ Entry Flag(ooct/common/trcommon.h)

TRANS_Set _ Entry _ Flagマクロは、変換テーブルエントリのフラグの1つ、JOIN、ENTRY 又はBUFFEREDをonにする。このマクロはロックプレフィックス(lock prefix)を持つアセンブリ言語命令を使用して、ビットを最小単位に(atomically)セットする。

【0053】TRANS Reset _ Entry _ Flag(ooct/common/trcommon.h)

TRANS_Reset _ Entry _ Flagマクロは、変換テーブルエントリのフラグの1つ、JOIN、ENTRY 又はBUFFEREDをoffにする。このマクロは、ロックプレフィックスを持つアセンブリ言語命令を使用してビットを最小単位にリセットする。

【0054】TRANS Entry _ FlagP(ooct/common/trcommon.h)

TRANS_Entry _ FlagPマクロは、変換テーブルエントリのフラグの1つ、JOIN、ENTRY 又はBUFFEREDの状態を読み取り、そして戻す。

【0055】TRANS Test _ And _ Set _ Entry _ Flag(ooct/common/trcommon.h)

TRANS_Test _ And _ Set _ Entry _ Flag手続きは、フラグの1つ、JOIN、ENTRY 又はBUFFEREDの状態を最小単位で読み取り、それが既にonされていなければonし、手続きをコールする前のフラグの状態に戻す。

【0056】TRANS Set _ Entry Address(ooct/comm

on/trcommon.h)

TRANS_Set_Entry_Address 手続きは、変換テーブルエントリのエントリポイントアドレスを書き込む。この手続きでは、ロックプレフィックスを持つアセンブリ言語命令を使用してアドレスを最小単位で書き込む。セグメントロックがない場合、エントリポイントアドレスはターゲット命令のアドレスであるが、セグメントロックがある場合には、SEGMENT_GATEデータ構造のアドレスであることに注意。

【0057】TRANS_Get_Entry_Address(ooct/common/trcommon.h)

TRANS_Get_Entry_Address 手続きは、変換テーブルエントリのエントリポイントアドレスを読み取り、そして戻す。セグメントロックがない場合、エントリポイントアドレスはターゲット命令のアドレスであるが、セグメントロックがある場合には、SEGMENT_GATEデータ構造のアドレスであることに注意。

【0058】セグメント

セグメントは、KOI システムが実行するコンパイルされたコードの単位である。以下に述べるコンパイラ104のマテリアルは、どの様にしてセグメントが生成され、そして削除されるかについて説明する。この項では、コンパイラ104はセグメントについて、どの様にインタプリタ110に告げるか、インタプリタ110はどの様にしてセグメントに入り、其処から出るか、及びコンパイラ104はインタプリタ110に対して、1つのセグメントの使用を停止し、他のセグメントに切り換えるよう、どの様に命令するか、について説明する。

【0059】セグメントが生成されると、インタプリタ110が入ることが可能な幾つかのASP命令アドレスがある。これらアドレスの各々に対して、コンパイラ104はセグメントへのエントリポイントを生成する。エントリポイントはセグメントに於ける特別な点であって、インタプリタ110は其処にジャンプすることができる。セグメント内の他の点では、コンパイルされたコードは或る値がレジスタにあると想定するので、其処にジャンプすることは安全ではない。これらエントリポイントが何処にあるかをインタプリタ110に知らせるため、コンパイラ104は各n番目のTRANS_Get_Entry_Addressについて、TRANS_Set_Entry_Addressをコールする。

【0060】インタプリタ110は、コンパイルされたコードがブランチロガー112に入る時、コンパイルされたコードに関してチェックを行う。コンパイルコードはTRANS_Entry_FlagPをコールし、現在のASPアドレスがエントリポイントを持っているかを調べる。もし持っていれば、TRANS_Get_Entry_Addressをコールして、アドレスを読み取る。セグメントロックがない場合には、コンパイルされたコードはセグメントをロックする(以下参照)。もしoffであれば、コ

ンパイルされたコードは、エントリポイントにジャンプする。コンパイルされたコードは、何時ブランチロガーから出るかを決定する。通常、この決定は、同じセグメントの部分ではない命令を実行する必要があるときに起こり、決定後、インタプリタ110にジャンプする。

【0061】コンパイラ104は、1つのコンパイルされたコードセグメントを削除し、もう1つ他のコンパイルされたコードセグメントを使用するよう、インタプリタ110に命令することができる。コンパイラ104は、変換テーブルエントリのENTRYビットをoffにし、エントリポイントアドレスを変更し、再度ENTRYビットをonにする。

【0062】セグメントロック

セグメントロックは、OOCTシステムの任意選択の機能である。システムの稼働と共にブランチロガー112はより多くの情報を得るから、コンパイラ104は古いセグメントよりも更に良い新たなバージョンのセグメントを作成することができる。コンパイラ104はセグメントロックによって、古いセグメントを新しいセグメントに換え、古いセグメントが使用したメモリを再生使用することができる。しかし、生憎なことに、セグメントロックはブランチロガー112及びコンパイルされたコードを低速化してしまう。そこで、OOCTコードを実行する時間と、それに使用するスペースとの間にはトレードオフ(交換)がある。この項では、セグメントロックの動作について説明する。

【0063】セグメントロックコードは2つの主要な部分を有している。第1の部分はセグメントロック実施部分を除くOOCTシステム全ての部分に対するインタフェースである。このインタフェースは、セグメントが4つの明確な状態の1つにだけ入ることができ、明確な方法でそれらの状態を変更することを保証する。第2の部分はセグメントロック自身の実施部分であって、上記インタフェースによる保証を満足する。

【0064】設計セグメントが取ることのできる状態はテーブル3に示されている。セグメントは到達可能か、又は到達不能かの何れかの状態を取ると共に、ロック又アンロックの状態を取る。セグメントは、変換テーブルに1つ以上のエントリポイントがあるとき到達可能であり、変換テーブルにセグメントへのエントリポイントがないとき到達不能である。エントリポイントは、ロック及び命令アドレスを含む構造である。1つ以上のインタプリタ110によって同時に使用されるロックは、幾つのインタプリタ110がエントリポイント及びそれを含むセグメントを使用したかをカウントする。セグメントの1つ以上のエントリポイントがロックされたとき、セグメントはロックされ、セグメントの全てのエントリポイントがアンロックされたとき、セグメントはアンロックされる。

【0065】コンパイラ104は、セグメントが到達不

能、且つアンロックの状態にあれば、セグメントの再生使用及び削除ができるが、セグメントが到達可能又はロックの状態にあれば、セグメントの再生使用はできない。コンパイラ104がセグメントを生成すると、全てのセグメントは状態U/Uにある。コンパイラ104が変換テーブルにエントリポイントを書き込むと、セグメントは状態R/Uに移動する。インタプリタ110のセグメントへの入出に応じて、セグメントは状態R/Lに移動し、また状態R/Uに戻る。コンパイラ104は、古いセグメントと同じ命令を変換する新たなセグメントを生成することができる。この場合、コンパイラは変換テーブルに古いセグメントのエントリポイントを上書きし、そのセグメントを到達不能とする。コンパイラ10

4がセグメントの最終エントリを上書きするとき、インタプリタ110がそれを使用している場合には、セグメントは状態R/LからU/Lに移動し、使用していない場合には、セグメントは状態R/UからU/Uに移動する。結局、セグメントを使用する全てのインタプリタ110はロックを解放し、セグメントは状態U/Uを取る。このとき、何れのインタプリタ110もセグメントを使用しておらず、そこにも入れないから、コンパイラ104はセグメントの再生使用もできるし、またセグメントの削除もできる。

【0066】

【表8】

テーブル8：セグメントが取りうる状態

状 態	到 達	ロック	状態の説明
U/U	否	否	何れのインタプリタ110もセグメントを使用せず、セグメントには入れない。コンパイラは何時でもセグメント削除できる。
R/U	可	否	1つを除く他のインタプリタ110はセグメントを使用していない。
R/L	可	可	1つ以上のインタプリタ110がセグメント及びその他を使用している。
U/L	否	可	1つ以上のインタプリタ110がセグメントを使用するが、その他を使用していない。

【0067】図6は、本発明の実施形態によって、インタプリタ110がセグメント122に入り、其処から出るまでを示す図である。図中央のセグメント122は、コンパイラ104によって作られたコードの単位である。セグメント122は、インタプリタ110によって使用される際には、常時ロックされていなければならない。従って、ロックカウンタ（図示せず）は、インタプリタがセグメント122にはいる以前には増分（加算）され、セグメント122から出た後は減分（減算）される。インタプリタ110は、エントリポイントをロックアップできず、エントリポイントを最小単位にロックするから、ロック後にエントリポイントが変化しなかったかを決定しなければならない。

【0068】図7は、コンパイラがセグメントを生成し、インタプリタ110によるセグメントへの到達を可能にし、古いセグメントへの到達を不能にし、そして古いセグメントを削除する方法を示す。ステップS200に於いて、コンパイラ104は新しいセグメントを生成し、関連するエントリポイントを変換テーブルに加える。エントリポイントがステップS200に加えられると、古いエントリポイントは書き換えられる。これによって、古いエントリポイントは到達不能となり、従ってタスク（例えば、インタプリタ110又はコンパイラ104）がそれをロックオン状態に保持していなければ、再利用できる。古いエントリポイントは再生使用リスト（図示せず）に書き込まれる。

【0069】ステップS202は、コンパイラ104がどの様に再生使用リストを使用するかを示す。ステップS202はエントリポイントがロックされているか否かをチェックする。エントリポイントがロックされていなければ、エントリポイントはどのインタプリタ110によっても使用されておらず、それ故、このエントリポイントを有するセグメントから取り除くことができる。しかし、そのセグメントが最早エントリポイントを持っていなければ、セグメントはタスク（例えば、インタプリタ110又はコンパイラ104）によって使用されず、如何なるタスクもそのセグメントには入れない。それ故、そのセグメントは削除することができる。

【0070】セグメントロックインタフェースは、OCTの殆どの部分が同期に関する詳細を無視することを許す。その理由は、セグメントは常に明確な状態にあり、そして全ての状態遷移は最小単位で起きるからである。しかし、セグメントロックコード内部では、インテルターゲットは、ハードウェアに於けるそうした複雑な動作を支持しないから、遷移は最小単位で起きてはいない。それ故、セグメントロックコードは遷移を自動的にさせる。

【0071】実 施

インタプリタ101及びコンパイラ104の実行手続きは、図6及び図7にそれぞれ図示されている。これら2つの手続きは、確実に遷移が自動的に行われるように共働する。以下の説明に於ける括弧内の参照番号は、図6

及び図7を参照する。

【0072】セグメントインタフェースの4つの状態の間には、6つの可能な遷移があり、それらは4つのグループに分かれる。第1の遷移は、コンパイラ104がセグメントのエントリポイントを変換テーブルに書き込むことによって、セグメントを到達可能にしたときのもので、U/UからR/Uへの遷移である(*6)。コンパイラ104は変換テーブルに書き込みができる唯一のタスクであるから、この遷移を自動的にするのに如何なる同期も必要ない。

【0073】第2の遷移グループは、R/UからU/Uの遷移と、R/LからU/Lへの遷移である。これらの遷移は、コンパイラ104がセグメントの最終エントリポイントを変換テーブルに上書きした時に起こる(*306)。コンパイラ104は変換テーブルに新たなエントリポイントを書き込むことができるが、インタプリタ110はエントリポイントを最小単位に読み取ることができず、それをロックする(*301、*302)。従って、インタプリタ110は、1つの動作でエントリポイントを読み取り、もう1つの動作でそれをロックしなければならない。もしインタプリタ110が、変換テーブルから古いエントリポイントを読み取り、次いでコンパイラ104が新たなエントリポイントを書き込み、そしてインタプリタ110が古いエントリポイントをロックするとしたら、これは潜在的問題を露呈することになる。この場合、コンパイラ104は、エントリポイントは到達不能であるが、インタプリタ110はセグメントに入ると想定する。これは誤りである。この問題を防止するため、インタプリタ110は変換テーブルがロッキング後、同じエントリポイントを含んでいるかをチェックする(*303)。変換テーブルが同じエントリポイントを含んでいれば、エントリポイントは到達可能であり、安全にセグメントには入れる。変換テーブルが同じエントリポイントを含んでいなければ、インタプリタ110はそのロックを解かなければならず、セグメントに入ってはならない。

【0074】第3の遷移グループは、R/UからR/Lへの遷移と、その反対のR/LからR/Uへの遷移である。前者の遷移は、インタプリタが変換テーブルからエントリポイントを読み込み、それをロックした時に起こる(*302)。後者の遷移は、インタプリタ110がセグメントのexit(*304)に於いてセグメントから離れて、アンロック手続き(*305)に移行するとき起こる。ロッキング及びアンロッキング命令は、それ自体セグメントには無く、セグメントは如何なる時もアンロック状態にあるから、コンパイラ104がそれを削除できる(*3011)ことは重要である。

【0075】第4の遷移は、U/LからU/Uへの遷移である。この遷移もインタプリタ110がセグメントを離れて(*304)、アンロック手続き(*305)に移行

するときに起こる。この遷移が起こった後、セグメントはアンロックされ、コンパイラ104は2つのテストを通り(*309、*3010)、そしてセグメントを削除する(*3011)。

【0076】インタプリタ110は任意の時間期間、セグメントのロックを保持できるから、コンパイラ104をそのロック時間待たせることは非効率である。それ故、コンパイラ104は、インタプリタ110がエントリポイントを使うのを防止するため、エントリポイントのロックを行わない。その代わり、コンパイラはセグメントを到達不能とし、後でロックが解放されたか否かをチェックする(*309)。一旦ロックが解かれれば、エントリポイントはフリーとなり、再利用することができる。

【0077】モニタメッセージ待ち行列
インタプリタ110は、コンパイラ104に対してシードアドレスを送る。このシードアドレス送信には、2つのメッセージ待ち行列が使用される。第1の待ち行列はシード送受信のため、KOI システムコール ScMsgSnd 及び ScMsgRcv を使用する。第2の待ち行列は、OOCTバッファの共用メモリ領域を使用する。この共用領域は、branch Seed Bufferと呼ばれる。

【0078】2つの待ち行列を使用する理由は、それら各々が1つの利点と、1つの不利な点を持っているからである。KOI システムコールは、インタプリタ110が使用するには費用が掛かり、そう頻繁には使用すべきではない。しかし、KOI システムコールは、コンパイルするシードがないとき、コンパイラ104がブロックすることを許す。このことは、KOI システムがコンパイラ104を使ってCPUが何か他の仕事をすることを許すことになる。共用メモリ領域の利点は、インタプリタ110がそれを非常に安価に使用できる点であるが、その不利な点はシードがない時、ブロックできない点である。

【0079】これら両方の待ち行列を用いることによって、OOCTはこれら2つの方法の利点を得ることができる。コンパイラ104が非動作状態にあるとき、ScMsgRcv をコールしてブロックする。この場合、インタプリタ110は ScMsgSnd コールを用いて次のシードを送り、コンパイラ104を起こす。コンパイラ104が動作状態にあるときは、高速のbranch Seed Buffer領域を介してシードを送る。コンパイラ104は1つのコンパイル動作を終了した後、sch Seed Buffer領域をチェックする。もし何か他にシードがあれば、コンパイラはそれらをコンパイルする。全てのシードのコンパイルが完了したとき、コンパイラは再度 ScMsgRcv をコールし、ブロックする。

【0080】V. インタプリタの修正 (EXECユニット)

OOCTの設計は、インタプリタ110に対する三種類の修正を含んでいる。その第1は、OOCTをインタプリタ11

0によって初期化できるようにするための修正、第2はインタプリタ110がブランチロギングを使用できるようにするための修正、最後はインタプリタ110がコンパイルされたコードへの遷移及び其処からの遷移ができるようにするための修正である。ここでは、これら修正の詳細について述べる。

【0081】OOCTのインタプリタコードは、2つのモード、OOCT_PERFORMANCE_MODE及びOOCT_DEBUG_MODEで実行することができる。ここでは、OOCT_PERFORMANCE_MODEの全ての特徴を説明すると共に、OOCT_DEBUG_MODEと何処が違うかについて述べる。

【0082】初期化
OOCTがコードをコンパイルし又はブランチをログする前に、インタプリタ110はOOCT_INITをコールして、OOCTデータ構造の初期化を行う。OOCT_INIT及びそれがコールする手続きは、以下のステップを実行する。

【0083】変換テーブルを初期化する。MCD命令はOOCTにシステムアドレススペースのページ数を告げる。手続きTRANS_Execution_Initは、システムページに関するエントリが、第2レベルの変換テーブルアレイを指すように第1レベル変換テーブルを生成する。これらアレイは、初期化の時にゼロにされる。変換テーブルについての更なる詳細は通信に関する項を参照。

【0084】ブランチロガー112を初期化する。手続きBRANCH_Execution_Initは、幾つかのデータ構造のためにOOCT_bufferのメモリを初期化する。これらの構造としては、第1にブランチ命令についてのプロファイル情報を含むブランチログ自身がある。第2には、ブランチロガー112を高速動作させるレベル1(L1)キャッシュがある。第3に、ブランチロガー112からコンパイラ104に送られるシードを含むシードバッファがある。第4に、コンパイルされたコードがコールする幾つかの大域機能(global functions)がある。これらのアドレスは、BRANCH_Execution_Initの実行中に、OOCT_bufferに記憶される。ブランチログ及びレベル1キャッシュに関する更なる情報については、上記ブランチロガー112に関する項を参照。

【0085】コンパイラ104のスタックメモリへの割当。コンパイラ104は、OOCT_bufferに割当られた特に大きいスタックを使用する。

1. コンパイラ104のゾーンメモリへの割当。コンパイラ104はコンパイル中、OOCT_bufferのこのメモリを使用する。
2. コンパイルされたセグメントメモリの割当。コンパイルされたコードはOOCT_bufferのこの領域に置かれる。
3. 統計的情報をゼロ化。OOCTが初期化される際、OOCTの統計的領域に於ける殆どの情報がリセットされる。

【0086】ブランチロガー
インタプリタを有するインタフェース

インタプリタ110がシステムコードで書いたブランチ命令を実行し、OOCTモードビットが設定されると、インタプリタ110は下記のルーチンの1つを介して、ブランチロガー112をコールする。

【0087】declspec(naked)OOCT_Log_Unconditional_Fixed_Branch()

ブランチを有するインタプリタによってコールされる。

増補: exc: ブランチ命令のアドレス

復帰: 復帰せず(IC_FETCH02へのジャンプのように作用する)

【0088】declspec(naked)OOCT_Log_Unconditional_Non_Fixed_Branch()

ブランチを有するインタプリタによってコールされる。

増補: exc: ブランチ命令のアドレス

復帰: 復帰せず(IC_FETCH02へのジャンプのように作用する)

【0089】declspec(naked)OOCT_Log_Conditional_Fixed_Branch_Taken()

ブランチを有するインタプリタによってコールされる。

増補: exc: ブランチ命令のアドレス

復帰: 復帰せず(IC_FETCH02へのジャンプのように作用する)

【0090】declspec(naked)OOCT_Log_Conditional_Fixed_Branch_Not_Taken()

ブランチを有するインタプリタによってコールされる。

増補: exc: ブランチ命令のアドレス

復帰: 復帰せず(IC_FETCH02へのジャンプのように作用する)

【0091】これら4つのルーチンは、変換先アドレスに対するコンパイルされたコードエントリポイントをチェックし、もし在ればそのエントリポイントにジャンプする。もしなければ、ルーチンは、branch_L1_Touch(次の項を参照)をコールして、ブランチログを更新し、インタプリタ110の取り出し(fetch)ルーチンへジャンプする。

【0092】ブランチログテーブル更新

図8は、本発明の好適実施形態によるBRANCH_RECORDの構造を示す。

【0093】ブランチロギングコードは、ブランチが何回実行したかをカウントする。ブランチロガー112がカウントの記憶に使用するデータ構造は2つ在る。その1つはブランチログであって、これは多重プロセッサシステムに於いて、全ての模擬的プロセッサによって共用される。もう1つは、多重プロセッサの各模擬的プロセッサに対する1つのレベル1(L1)キャッシュである。ブランチ実行カウントは、先ずキャッシュに書き込まれ、次いでブランチログに書き込まれる。この項ではL1キャッシュ及びブランチログの構造について説明する。また、ブランチロガー112がどのようにしてそれらを使用するかについても説明する。

【0094】各ブランチに関する情報は、BRANCH RECORDと称する構造に記憶される。この構造は、ブランチのアドレス、ブランチの変換先、ブランチに続くフォールスルー(fall through)命令、ブランチが実行した大凡の回数、及びブランチが取られた(taken) 大凡の回数を含む。BRANCH RECORDの最終フィールドは、もう1つ他のBRANCH RECORDに対するポインタであって、連結リストにBRANCH RECORDを接続するのに使用される。

【0095】ハッシュテーブルは、連結リストの配列として構成される。

【0096】図9は、ブランチログの構造を示す。それはBRANCH RECORDを記憶する大きなハッシュテーブルである。各インタプリタ110は、可変 local_branch_header_table 自身のコピーを持っているが、それら全てはOCTバッファ領域に於ける同じ配列を指す。local_branch_header_table の要素は、BRANCH RECORDのリストに対するポインタである。ブランチに関するBRANCH RECORDの探索手続きは以下の3ステップである。

1. 変換先アドレスをハッシュする。(index =BRANCH_HASH(destination_address)% BRANCH_TABLE_SIZE)

2. リストのヘッドを取得する。(list =local_branch_header_table[index])

3. 同じブランチアドレスを持つ記録が見つかるまで探索する。(while(list->branch_address!=branch_address)list = list->next)

【0097】図9は特に、可変 local_branch_header_table が、リストに対するポインタ配列であることを示している。各リストは同じ変換先アドレスを有するBRANCH RECORDを含んでいる。リストがない場合、local_branch_header_tableのポインタは空である。

【0098】ブランチログはブランチに関する情報の全てを含んでいるが、2つの問題を抱えている。その1つは、BRANCH RECORDの探索及び挿入動作が遅い点である。インタプリタ110が常時ブランチの記録を取るには遅すぎる。もう1つの問題は、全てのインタプリタ110が同じブランチログを使用する点である。BRANCH RECORDのリストを一貫して矛盾無く保つために、一度に1つのExecだけしかブランチログにアクセスできない。このことは、単一のプロセッサシステムの場合はもとより、多重プロセッサシステムの動作を更に減速させる。この問題を解決するために、各インタプリタ110には、1つのL1キャッシュがある。このL1キャッシュには高速でアクセスができると共に、インタプリタ110はこれと平行に、それらのL1キャッシュにアクセスすることができる。各L1キャッシュは、BRANCH_L1_RECORD構造の2次元配列として構成される。この配列のベース(基底、基準)アドレスは可変branch_L1_tableに記憶される。

【0099】図10はL1キャッシュの構造を示す。L1キャッシュは、BRANCH_L1_RECORDの2次元配列である。第1次元はBRANCH_L1_SETS(現在32)であり、第2次元はBRANCH_L1_SETSIZE(現在4)である。配列の各行は1つのセットである。同じブランチ命令は、常にキャッシュの同じセットを使用するが、それは異なる場所に於いても可能である。

【0100】図10に示すように、L1キャッシュは複数のセットで構成される。ブランチに対するセット数は、(branch_address + branch_destination)% BRANCH_L1SETSに等しい。セットの4つの要素は同じセット数の4つの最新ブランチを保持する。これは4-ウェイセット連想法(4-way set associativity)と言われる。同じセット数を有し、ほぼ同時に実行される幾つかのブランチが在るとき、キャッシュの性能は改善される。

【0101】図11は、本発明の実施形態によるインタプリタ110によって、L1キャッシュの動作を実行させる方法を示す。即ち、図11はL1キャッシュを用いたブランチロギング方法を示す。

【0102】最適化オブジェクトコード変換方法は、コンパイル未了ブランチを記録するのに、2つの形のメモリを利用する。即ち、

1. 記録されたブランチの数に比例する動的変換サイズを有するブランチログ、及び

2. 限られた数のコンパイル未了の記録されたブランチが、アクセスを高める順序に従って記録される、L1キャッシュと称するブランチキャッシュ、を利用する。これらブランチログ及びL1キャッシュは、オペレーティングシステムによって管理される仮想記憶位置を表す。それ故、ここでの用語“L1キャッシュ”は、コンパイル未了ブランチを記憶するキャッシュに任意に与えられる用語であって、一般に、Pentium Pro等のプロセッサに見られる“L1キャッシュ”と混同してはならない。

【0103】本発明による最適化オブジェクトコード変換プログラムは、インタプリタ110が複数の異なるブランチルーチンをコールすることができるようにする。しかし、各ブランチロギングルーチン自身は、コンパイルされたコードヘジャンプするか、又はブランチ命令を記録するかを決めるサブルーチンをコールする。図11は、このサブルーチンを詳しく示している。

【0104】上記説明を考慮しつつ、L1キャッシュを使用したブランチロギング方法を実行するため、先ずステップS400からこの方法を始める。ステップS401において、インタプリタ110は、ブランチの変換先に対するコンパイルされたコードのエントリポイントをチェックする(即ち、問題のセグメントが事前にコンパイルされているか否かをチェックする)。もしエントリポイントが在れば、即ち“yes”ならば、コンパイルされたセグメントが在り、直ちにコンパイルされたコードセグメントを実行するためステップS402ヘジャンプ

する。このコンパイルされたコードセグメントの実行は、エンドフラグに到達するまで続けられ、これが終了すると、次のセグメント実行に戻る。勿論、ブランチが既にコンパイルされているので、ブランチはブランチログに記録されない。

【0105】ステップS401で、エントリポイントがなければ、即ち“no”ならば、ブランチ命令に対応するコンパイルされたセグメントはなく、ステップS404に移り、インタプリタ110はL1キャッシュを調べて、L1キャッシュに記憶された複数のブランチに一致するブランチがあるか否かを決定する。

【0106】ステップS404は、L1キャッシュに記憶された複数のブランチに一致するブランチがあるか否かを決定する。L1キャッシュは複数のセットに分かれており、各セットは固有のセット番号によって指定されている。本発明の実施形態では、各セットは4つのブランチを含んでいる。

【0107】ステップS404は、最初に現在のブランチアドレスに対応するキャッシュセットの数“S”を決める。ここで、 $S = (\text{branch_address} + \text{branch_destination}) \% \text{BRANCH_L1_SETS}$ である。次に、branch_L1_table[S]の各要素は、現在のブランチのアドレス及び変換先に関して順次チェックされる。もし一致するブランチが検出されると、即ち“yes”ならば、ステップS406に進み、フィールド“encountered_sub_count”（何回ブランチに出会ったかを示すフィールド）及び“taken_sub_count”（何回ブランチが取られたかを示すフィールド）が更新され、次にステップS407に進む。

【0108】ステップS407では、現在のブランチアドレスが、所定の閾値より大きいかが決定される。好適な閾値は1000ビットのオーダーである。こうして、フィールド encountered_sub_count は、ステップS407で閾値と比較される。もしそれが閾値を越えていれば、即ち“yes”ならば、ステップS410に進み、このブランチに関するキャッシュされた情報は、ブランチログに書き戻される。他方、閾値を越えていなければ、即ち“no”ならば、ステップS412に進む。ステップS412は、IC-FETCH02へジャンプする現在のサブルーチンの終わりであり、即ちインタプリタ110のエントリポイントである。

【0109】もし正しいブランチがキャッシュになれば、即ちステップS404で“no”ならば、ステップS408に進んで、上記“S”によって指定されたセットの1つBRANCH_L1_RECORD（即ち、encountered_sub_count、及び taken_subcount 等の更新可能な全てのフィールドを含む記録）はL1キャッシュから除かれ、ブランチログに書き込まれる。次に、現在のブランチ情報は“S”によって指定されるセットに書き込まれる。更に、現在のブランチ記録をセット“S”に書き込み

中、この現在のブランチ記録はセットの第1要素として置かれる。これは、同じブランチが再度実行される可能性があり、これによってシステムの性能及び効率が上がるからである。言い換えれば、ステップS404は高速で実行される。ブランチがキャッシュに在る時でさえ、即ち“yes”の場合、もしそのブランチが何回も何回も実行されたとしても、最終的にフラッシュされるので、そのブランチをブランチログにコピーしておくこともできる。

【0110】L1キャッシュを使用する際、ステップの順序は大凡常に、S400、S404、S406、S407、そしてS412の順である。従って、本発明はそれらのステップを出来るだけ高速にすることを求める。現在のブランチ情報がセットの第1要素に入力されると、インタプリタ110は同じブランチを再度実行するようになるから、そのブランチ情報がステップS404を高速化する。

【0111】上記のブランチロギング方法は、前もってコンパイルされたコードを実行し、そしてコンパイルの閾値レベルに達していないブランチ命令へのアクセスを高めることによって、プロセッサへの負担を軽減する。この点について、OOCの主旨は、ステップS400が殆ど常に分岐“yes”を取るようにすることである。ブランチを頻繁に実行する場合には、その変換先に関するコンパイルされたコードセグメントがなければならない。

【0112】第2の目的は、ステップS401に続く経路“no”を高速化し、まだコンパイルされてないブランチがプログラムの実行速度を目立って落とさないようにすることである。この経路“no”の最も遅い部分を“フラッシュ”と呼ぶ。両ステップS408及びS410では、ブランチ情報はL1キャッシュから“フラッシュ”され、ブランチログに書き込まれる。シードをコンパイラに送るには、ブランチ情報をフラッシュすることが必要になり、このことがコンパイルされたコードを発生させ、ステップS400がやがて、このブランチに対し“yes”と応えるようにさせる。

【0113】しかし、コンパイル未了のブランチアドレスを実行するたびに、ブランチの情報をフラッシュする必要はない。100回乃至はそれ以下の実行毎にフラッシュを掛ければ良い。それ故、本発明はフラッシュを含まないステップS400、S404、S406、S407及びS412の実行速度を上げることを求める。従って、二者択一状態が起こらない限り、より高速な経路を常に選択するようにする。ステップS404では、ブランチ情報がセットに見つからない可能性がある。そこで、ステップS408への経路“no”を取る。ステップS407で、もしブランチが“閾値”回数以上に実行されるようであれば、フラッシュを含むステップS410へ経路“yes”を取る。

【0114】OOCT_DEBUG_MODEでも、L1キャッシュ方法が使用されるが、キャッシュをフラッシュするための閾値は1にセットされ、情報はブランチの実行毎にブランチログに書き込まれる。これは、OOCT_DEBUG_MODEを更に遅くする。

【0115】シード選択

ブランチ命令が頻繁に実施されるとき、ブランチローガー112は、その変換先アドレスをコンパイラ104に送る。このアドレスは“シード”と呼ばれ、そしてシードの選択はOOCTシステムの非常に重要な部分である。

【0116】シードは、手続きの開始又はループのヘッドに於けるアドレスでなければならない。それ故、ブランチローガー112は無条件ブランチの変換先であるシードを送るだけである。シードは頻繁に実行されるアドレスでなければならない。従って、encountered count フィールドが閾値より大きい場合にだけ、ブランチの変換先がシードとなる。閾値は、seed production threshold と称するフィールドのOOCTバッファに記憶される。閾値は時間を切り換えることができる。これについては、次の項で説明する。

【0117】閾値設定

シードを送るか否かを決定するのに固定閾値を使用することについては、2つの欠点がある。その第1は、コンパイラ104が遊休状態(idle)にある間、固定閾値が高すぎる嫌いがある。この場合、コンパイラ104には為すべき有効な作業があるが、ブランチローガー112はコンパイラ104に対して、何をすべきかを指示しない。また、第2として、メッセージ待ち行列が一杯の状態にある間、固定閾値が低すぎる嫌いがある。この場合、ブランチローガー112は、たとえシードが待ち行列に適合しなくても、シードを送ろうとする。これは時間の浪費に繋がる。

【0118】幸いなことに、これら2つの状態、即ちコンパイラ104の遊休状態及びメッセージ待ち行列の満杯状態は検出することができ、そして閾値を変えることができる。ブランチローガー112は、OOCTバッファのnum_monitor_seed_messagesと称するフィールドを読み取ることによって、コンパイラ104が手続きbranch_update_entry に於いて遊休していることを検出する。このフィールドが0の場合、コンパイラ104は、送られてきた全てのシードを完了している。閾値が高すぎると、ブランチローガー112はそれを下げる。ブランチローガー112がシードを送ろうとして、手続きbranch_send_seedで満杯のメッセージシードを検出すると、ブランチローガーはメッセージが送れなかったことを示すエラーコードを受け取る。閾値が低すぎると、ブランチローガー112はそれを上げる。

【0119】OOCT_DEBUG_MODEでは、閾値は決して変化しない。この場合、その値はOOCTINIT手続きの第3引数にセットされる。

【0120】多重タスキング処理

OOCTは複数のインタプリタ110を有する多重プロセッサ上で作動する。これらのタスクは個々にブランチL1キャッシュを有しているが、同じブランチログテーブルを使用する。ブランチ情報がL1キャッシュからブランチログテーブルにフラッシュされると、インタプリタ110は、如何なる他のExecとも競合しないログをテーブル上で入手する。ブランチログロックに関するコンテンツン(競合)を扱うには2つの方法がある。第1の方法は、ロックが使用可能になるまでインタプリタ110を待機させ、次いでロック使用が可能になったら、そのブランチ情報を書き込む方法である。この方法は、インタプリタ110の実行速度をより遅くするが、ブランチログをより正確にする。また、第2の方法は、もしインタプリタ110がロックを得られなければ、ブランチ情報を書き込まずにあきらめる方法である。この方法は、インタプリタ110を高速にするが、或るブランチロギング情報を失うことになる。インタプリタ110のスピードは、ブランチログの正確さより重要なので、OOCTは第2の方法を使用する。ブランチログ情報は、システムがうまく機能する程度に大凡の正確さを持っていることが必要なだけである。

【0121】OOCTが複数のインタプリタ110と共に動作する際、複数タスクの1つは、OOCTバッファ及びブランチロギングデータ構造を初期化するために、OOCT_INITをコールする特別なマスタートaskである。その他のタスクは、幾つかのローカル変数及びそれらのブランチL1キャッシュを初期化しなければならないだけの従属タスクである。従属タスクは、マスタートaskがOOCT_bufferの初期化を完了した後に、SlaveOOCT_Initをコールする。マスタートaskと従属タスクとの間の同期を取るには下記の方法を使用する。即ち、

【0122】マスタートaskに対する方法

1. MCD命令を実行してOOCTをonする。
2. OOCTバッファ及びブランチロギングデータ構造を初期化するOOCT_INITをコールする。
3. 従属タスクを作動させる。
4. インタプリタへジャンプする。

【0123】従属タスクに対する方法

1. 休眠状態に入る。マスタートask(上記ステップ3)の実行により作動開始。
2. タスク個々のブランチL1キャッシュを初期化するSlaveOOCT_Initをコールする。
3. インタプリタへジャンプする。

【0124】ユーザ/システムスペース変換

OOCTシステムはASPアドレススペースのシステムページからの命令をコンパイルするだけであって、ユーザページは無視する。インタプリタ110の個々の領域のOOCTSTSビットは、ブランチローガー112をコールするかどうかを制御する。このビットは、二つのマクロNEXT_CO

及びNEXT_QUN によって主として制御される。しかし、OOCTコードがこのビットをセットしなければならない場合が1つある。コンパイルされたコードセグメントが非固定ブランチ命令で終わるとき、OOCTコードは PSW_IA をシステムスペースからユーザスペースに移動させ、OOCTSTS を0にセットする。それ故、非固定ブランチ命令で終わるコンパイルされたコードセグメントは、変換先アドレスをチェックすると共に、OOCTSTS ビットを正しくセットするルーチンbranch_Exit_Log にジャンプする。

【0125】コンパイルされたコードインフェース、コンパイルされたコードへからの遷移
インタプリタ110がブランチロガールーチンをコールするとき、実行をコンパイルされたコードに移行し、ブランチの変換先向けにコンパイルされたセグメントを見つける (図11参照)。セグメントロックキングがoff

の時、インタプリタ110はエントリポイントへ直接ジャンプする。セグメントロックキングがonの時、インタプリタ110は、エントリポイントへジャンプする前に、セグメントのロックを試さなければならない。セグメントをロックできたら、インタプリタ110はエントリポイントへジャンプする。もしできなければ、インタプリタ110へジャンプバックする。

【0126】制御がコンパイルされたコードセグメントから出るための実行方法には幾つかある。これをテーブル4に示す。これら全ての場合、制御がインタプリタ110に戻るとき、ESI及びEDIレジスタは正しい値を持ち、インタプリタ110の個々の領域は完全なK状態を持っている。

【0127】

【表9】

テーブル4：制御がコンパイルされたコードセグメントから出る方法

最終K演算コード	コンパイルされたコードが行うこと
固定ブランチ、又は直線的K演算コード	宛先アドレスがコンパイルされたエントリポイントを持っているか調べる。もし持っていれば、インターセグメントをエントリポイントへジャンプさせる。持っていないければ、制御をIC_FETCH02 に於いてインタプリタ110に戻すか、又はセグメントロックキングがonの時、branch_Exit に戻す。
非固定ブランチ	OOCTSTS ビットをセットするbranch_Exit_Logへジャンプし、PSW_IAがまだシステムページにあれば、ブランチロガー112をコールする。
LPSW, SSM, STNSM, MCD, CALL, RRTN, SVC, MC, BPC, LINK, LINKD, LOAD, LOADD, DELT, DELTD, FBPC	セグメントロックキング無し：IC_FETCH02にジャンプして演算コードを実行。セグメントロックキングあり：branch_Exit_Interpretへジャンプ。
RISCモードへ切り換えるSAM演算コード	セグメントロックキング無し：IC_FETCH02にジャンプしてSAM演算コードを実行。セグメントロックキングあり：branch_Exit_Interpretへジャンプ。

【0128】セグメントロックキングがonの時、インタプリタ110はコンパイルされたコードを実行する間、コンパイルされたコードセグメントのロックを保持する。インタプリタ110はセグメントを出た後、このロックを解除しなければならないから、コンパイルされたコードは、ロックを解除し、インタプリタ110へジャンプするブランチロガー112の幾つかの手続きをコールする。

【0129】割り込み
コンパイルされたコードの実行中にできる割り込みには、入出力(I/O)割り込み、又はMCHK割り込み等の幾つかのものがある。コンパイルされたコードは個々の領域のINTINFフィールドをチェックして、割り込みが起こったかを検出する。コンパイルされたコードはできる限り無限ループのこのフィールド内部をチェックし、常に確実に割り込みを見逃さないようにする。割り込みが起きた場合には、コンパイルされたコードは、完全なK状態に

あるインタプリタ110のルーチンIU_OINTCHK をコールする。それはインタプリタ110がコンパイルされたコードに戻ることを期待する。

【0130】インタプリタコールバック

K演算コードの或るものは、OOCTによって変換されない。その代わり、コンパイルされたコードはインタプリタ110のサブルーチンIC_OOCTをコールし、その演算コードを変換してコンパイルされたコードに戻す。コンパイルされたコードは、IC_OOCTをコールする前に、ESI及びEDIレジスタが正しい値を有すること、及び個々の領域が完全なK状態を有していることを確認する。

【0131】インタプリタ110は、サブルーチンIC_OOCTを実行中にエラーを検出した場合には、手続きOOCT_EXCPをよびだし、コンパイルされたコードには戻らない。セグメントロックキングがonの場合、OOCT_EXCPはセグメントロックを解除する。

【0132】例 外 (異常)

変換された演算コードが、マスクを外した(unmasked)例外、例えば演算異常、即ちゼロ除数異常等を有している場合、コンパイルされたコードはインタプリタのサブルーチンIC_PGMxx をコールする。ここで、xxは01hと21hとの間のエラーコード番号である。インタプリタ110は例外を処理して、戻ろうとする。インタプリタ110が戻れない場合、如何なるセグメントロックも解除するOOC、EXCPをコールする。

【0133】大域機能の利用

K演算コードの或るもの、例えば文字処理用演算コードは、多数のターゲット演算コードに変換される。これら演算コードの多重変換は、コンパイルされたコードがこれら演算コードを実行するためにコールする大域機能と称するサブルーチンに関するセグメントメモリを多量に使用する。これら大域機能は、それらがコンパイルされたコードからコールされ、コンパイルされたコードに戻るよう特別に書かれている点を除けば、丁度、K演算コードを実行するインタプリタ110のサブルーチンのようなものである。5つの演算コードSBE、CC、MV、TS、及びCについて大域機能がある。実験によれば、大域機能はインタプリタ110のIC、OOCエントリポイントをコールするよりも遥かに高速であり、また演算コードをターゲット命令に複数回コンパイルする場合に較べて、遥かにメモリの使用量が少なくなることが示されている。

【0134】VI. コンパイラ

概観

コンパイル動作の詳細について調べる前に、コンパイラ104の主要目的及びその構造を十分に理解することが重要である。コンパイラ104の目的は、現在実行中のプログラムの多量に実行された部分を最適化ターゲットコードに変換し、実行に当たって、インタプリタ110がこのコードを利用できるようにすることである。

【0135】図12はコンパイラ104の全体構造を詳しく示す。コンパイラ104は上記ブランチローガー112からシードを受け取り、コンパイル動作を開始する。このシードは、現在実行中のプログラムに於いて多くのブランチのターゲット(目標)であったオリジナル命令のアドレスである。このことは、現在実施中のプログラムの多量実行部分を探索するための始点を与える。ブロックピッカ114はこのシードを、ブランチローガー112によって与えられるその他の情報と併せて使用し、コンパイルされるプログラム部分を取り出す。

【0136】コンパイル対象のオリジナルコードが選択されると、それは3つの主要段階を経て実行される。第1段階では、K演算コードを、コンパイラ104の残りによって使用される中間言語(IL)に変換する。中間言語は、IL発生器124によって発生される。第2段階では、ILに関する種々の解析及び最適化変換が上記

の最適化方法を使用して実施される。この段階は、オプティマイザ(最適化器)126として図示されている。第3段階、即ち最終段階では、ILを再配置可能なマシン語に変換する。この段階は、最適化コード発生ユニット118として図示されている。

【0137】コンパイラ104の最終の仕事は、インタプリタ110が最適化コードを利用できるようにすることである。セグメントデータ構造は、セグメント導入ユニットを用いて、最適化コードのコピーによって生成される。次いで、セグメントはOOCバッファ(図示せず)内の共用領域に導入される。変換テーブルは最終的には更新されるので、コンパイルされたKコードに対するインタプリタ110による如何なるブランチも、この新たなオブジェクトコードを代わりに使用する。

【0138】この項の残りの部分では、上記コンパイラ104のそれぞれの段階を追って詳しく説明する。また、その他種々多くの実施段階の詳細については、この項の終わりで説明する。

【0139】ブロックピッキング

コンパイラ104は単一シードアドレスを受けて、コンパイル動作を開始する。コンパイラは手続き本体らしきものを読むまで、オリジナル命令を読み取る。次いで、コンパイラは104は次の段階であるIL発生器に、このオリジナル命令セットを送る。コンパイラ104が読んだ命令単位は、基本ブロックと呼ばれ、それ故、この段階をブロックピッカ、即ちブロックピッカ114と言う。

【0140】この基本ブロックは、制御が最初の命令にしか入れず、また最後の命令からしか出られない一連の命令である。このことは、最初の命令だけがブランチのターゲットであって、最後の命令だけがブランチ命令であることを意味する。また、このことはブロックの最初の命令が実行されると、全ての命令が実行されることを意味する。

【0141】ブロックピッカ

図13は本発明の一実施形態によるブロックピッカ114の一例を示す。手続きOOC_ParseFrom はブロックピッカ114を具体化する。ブロックピッカは、一度に1つの基本ブロックを読み取る。基本ブロックは次の5つの理由の1つに該当するとき終了する。

1. もしパーザ(parser: 構文解析系)がブランチ命令を読み取ると、基本ブロックはそのブランチで終了する。
2. もし次の命令が既に構文解析されていると、K演算コードは、1つのセグメントに一度しか現れないから、基本ブロックは現在の命令で終了する。
3. もし次の命令が結合点であれば、結合点は基本ブロックの最初になければならないから、基本ブロックは現在の命令で終了する。
4. もし現在の命令が演算対象となる因子(ファクタ)

でって、命令の代わりにデータを伴うことが出来ると、基本ブロックは現在の命令で終了する。

5. もし現在の命令が違法命令であれば、基本ブロックは現在の命令で終了する。

【0142】ブロックピッカ114は各ブロックを読ん

テーブル5：ブロックを読んだ後の動作

現在ブロックの終了点	ブロックピッカ114の動作
条件付きブランチ	フォールスルー命令及びブランチ宛先命令に於いて構文解析を継続。
無条件固定ブランチ	ブランチ宛先命令に於いて構文解析を継続。
非固定ブランチ	ブランチ宛先未知のため構文解析を停止。
終了命令因子又は違法命令	次のバイトが命令を構成しないから構文解析を停止。
他の命令	フォールスルー命令での解析を継続。

【0144】一例を図13に示す。ブロックピッカ114はシード命令、即ちLB命令で動作を開始する。その命令は終了命令のブランチ又は因子でもないので、ブロックピッカは次の命令に進む。その命令は、条件付きブランチであるTH命令である。ブロックピッカ114は、それが条件付きブランチであるため、それを読むのを停止する。次に、ブロックピッカはLH及びLF両命令に於いて新たなブロックの読取りを継続する。ブロックピッカがSVC命令を読むと、SVCは終了命令の因子であるから、そのブロックを終了する。ブロックピッカはGO命令を読むと、GOはブランチ命令であるから、そのブロックを終了する。L8命令はブランチ変換先命令であるから、ブロックピッカはL8命令に於ける読みを継続する。ST8命令を読んだ後、ブロックピッカ114は次の命令を既に読んであるので、そのブロックを終了する。

【0145】ブロックピッカ114が読む命令の数には上限がある。この制限の目的は、コンパイラ104が変換元命令のコンパイルの間に、メモリを使い果たすのを防止するためである。この制限は、OOCt_trace.cの定数MAX_KINST_NUMによって設定される。現在の例では500に設定されている。

【0146】ブロックピッカ114が命令を読もうとすると、ページフォールトが起こることがある。もしこのページフォールトが起きると、ブロックピッカ114は現在のブロックを読むことを停止しするが、まだ試していない何れかのブランチ変換先から読む動作を継続する。このことは、たとえブロックピッカが、シードから到達できる全ての命令を解析できない場合でも、コンパイラ104がセグメントを生成することを可能にする。

【0147】ブロックレイアウト

ブロックピッカとなる基本ブロックを選択した後、手続きOOCt_GenerateILをコールし、コンパイラ104の残りが使用するIL命令を生成する。この時、ブロックの

後、そのブロックの終わり方によって、次を取るべき動作を決める。以下、これら可能な動作をテーブル5に示す。

【0143】

【表10】

順序を再配置することが出来る。これはブロックレイアウトと呼ばれ、フォワード条件付きブランチを取らない場合には、Pentium Proは高速で走行するから、コンパイラ104が、Pentium Proプロセッサにとって更に良いコードを作成する助けとなる。

【0148】図13の例について考えてみる。この例は1つの条件付きブランチ、即ちTH命令を有している。オリジナル命令に於いて、フォールスルー基本ブロックはLHで始まるブロックであり、変換先ブロックはLFで始まるブロックである。条件付きブランチが時間の75%を占める場合、LF基本ブロックをフォールスルー位置に置き、LH基本ブロックをブランク取り込み位置に置けば、条件付きブランチは更に速く走行する。

【0149】OOCt_GenerateILはブランチログの情報に従ってブロックを配置する。OOCt_GenerateILは可能な限り、条件付きブランチの最も一般的なサクセサをフォールスルー位置に置いて行く。この手続きによって、コンパイラ104の最適化段階に渡されるIL命令のリストが作成される。

【0150】中間言語(IL)の発生

この項では、K演算子に関するコンパイラ104の中間言語(IL)表現の発生過程を説明する。ILが如何に発生されるかを直接説明する前に、ILの概要を述べ、ILの理解に重要なデータ構造を説明する。

【0151】ILの概要

コンパイラ104の主な解析及び変換パスは、特別なマシン独立命令セットである中間言語を用いて実施される。中間言語の使用は、2つの主な理由から、標準的なコンパイラ104の技術である。その理由の第1は、ILが解析及び変換を簡単にする典型的な構造を有していること。その第2は、ILによって多くの異なる変換元言語が同じ最適化及びコード発生段階を用いることができ、また異なるプラットフォームに対する再対象化を容易にするからである。

【0152】OOCTが使用するIL（以下、単にILと言う）は、ここではテーブル6に掲げた40の演算コードから構成される。これらの命令は3つの主なカテゴリに分類される。その第1は、標準マシン演算コードへの簡単なマッピング機能を有するADD及びLOAD等の機能演算コードである。第2は、LABEL及びCGOTO等の制御の流れを扱う演算コードである。そして最後は、バックエンドによって発生されるコードに直接に対応しない、コンパイラ104が特別なマークとして使用する多くの特別コードである。これら特別マークコードについては、別の項で説明する。ILは仮想マシン（仮想計算機）を表すから、更に機能が要求される場合はILに対する他の演算コードの追加は簡単である。

【0153】ILは命令から構成され、それぞれ命令は演算コードの1つ、形式、及び多数の擬似レジスタ引数を特定する。コンパイラ104が支持する形式は符号付及び符号なしの8ビット、16ビット及び32ビット値である。SET演算コードによって用いられる即時値、

テーブル6：IL演算コード

演算コード	説明
LABEL	ジャンプ動作のターゲットとなる場所をフローグラフにマーク
GOTO	labelへジャンプ
CGOTO	擬似レジスタのプール値に基づいて、labelへ条件付きジャンプ
IGOTO	擬似レジスタの値によって決まるアドレスに間接ジャンプ
SET	擬似レジスタに即時値を入力
ASSIGN	1つの擬似レジスタの値を他の擬似レジスタに移動
OASSIGN	エイリアシングを明確にするための、擬似レジスタが何処で重なっているかを示す特別マーク命令
CVT	擬似レジスタを1つの形式から他の形式（例えば、符号延長、ゼロ延長）に変換
NEG, CMPL, BSWAP	単項否定、論理補数、バイト交換
ADD, SUB, MUL, DIV, REM	2進加算、減算、乗算、除算、剰余
ASL, ASR	演算シフト（左、右）
LSR	論理シフト（右）
BAND, BOR, BXOR	2進論理AND、OR、XOR
HQ, NB, LT, LE, GT, GE	2つの入力オペランドを比較し、 $op1 == op2$, $op1 != op2$, $op1 < op2$, $op1 <= op2$, $op1 > op2$, $op1 >= op2$ ならば、出力オペランドを真にセットする
TESTZ, TESTNZ	2つの入力オペランドを比較し、 $(op1 \& op2) == 0$, $(op1 \& op2) != 0$ ならば、出力オペランドを真にセットする

及びLOAD演算コードによってメモリからロードされる値は別として、全ての引数は擬似レジスタに渡される。この擬似レジスタは、単にILマシンのレジスタである。コンパイラ104は任意の数の擬似レジスタを許容し、これらレジスタはそれぞれ所定のサイズ（例えば、16ビット）を有している。各擬似レジスタは特定の記憶場所に直接対応している。OOCTに関して、これら記憶場所は、個々の領域のOOCT特別部分にある。記憶場所への擬似レジスタのマッピングには2つの利点がある。その第1は、ILを連続的に流す。共通使用の値を一時的にロードし、それらをメモリに戻して記憶する、と言ったIL動作は必要がなくなる。第2に、コンパイラ104は共通使用の値をマシンレジスタに頻繁に割り当てることが出来るので、余分なロード又は記憶動作を省くことが出来る。

【0154】

【表11】

【0155】

【表12】

テーブル 6 (続き)

演 算 コード	説 明
CMP	2つの入力オペランドを比較し、出力オペランドを、 $op1 < op2$ ならば -1 に、 $op1 = op2$ ならば 0 に、 $op1 > op2$ ならば 1 に、セットする。これは OOC にによって現在使用されない。
LOAD	特別記憶場所から値を擬似レジスタにロード
STORE	擬似レジスタの値を特別記憶場所に記憶
GCALL	所定の大域機能セットの 1 つに対し機能コールを実施
ICALL	IGOTO と同様な間接機能コールを実施
EXIT	コンパイルされたブロックから出る (OOC は現在使用せず)
ENTRY	制御がフローグラフには入れる点をマーク
SYNC	擬似レジスタをメモリに対してフラッシュする点をマーク
EXTMOD	外的に修正される擬似レジスタをマーク。機能コールによる擬似レジスタの修正を取り扱うのに使用
SBTCC	演算に基づく条件コードの値にブール値を設定。フラグを用いる場所を表すのに使用。現在、全ての SBTCC 演算はサクセサに折り込まれ、その演算は発せられないが、SBTCC の使用によって、コンパイラ 104 を要せずに条件コードの値の流れを明確にし、単一 IL 演算に対し複数の宛先を表す。

【 0156 】 特別 IL 演算コード

OOC の IL は特別なオブジェクトを持った幾つかの演算コードを含んでいる。殆どの IL 演算コードは、バックエンドで発生されるコードに対応する。その代わり、これら特別命令は、何か特別なことが起きたことをコンパイラ 104 に知らせる信号として作用する。IL は次の特別演算コード、ENTRY、SYNC、EXTMOD、及び OASSIGN を含んでいる。この項では、これら演算コードの最初の 3 つについて説明する。OASSIGN については、上記テーブルで十分に説明されている。

【 0157 】 ENTRY 演算コードは制御がフローグラフに入れる点をマークする。OOC にによって発生されるコードは、外部結合点を表す複数の外部エントリポイントを有することができる。各外部エントリポイントは、対応する ENTRY IL 命令を有している。ENTRY 命令はコードの終わりで発生し、その直ぐ後にコードの主体内のラベルにジャンプする GOTO 命令が続いている。ラベルに直接ジャンプする外部エントリジャンプを持つ代わりにエントリを使用する理由は、ENTRY とラベルへのジャンプとの間に、コード発生器によるフィル (fill) 挿入を可能にするためである。

【 0158 】 図 14 は、ENTRY 命令と GOTO 命令との間に、フィルを挿入した 2 つの外部エントリポイントを持つコードの概略を示す。即ち、図 14 は本発明の実施形態によるエントリの例を特に示した図である。

【 0159 】 SYNC 演算コードは、擬似レジスタの 1 つの範囲がメモリにフラッシュされたことを保証するのに用いる。特に、OOC は SYNC を用いて、インタプリタ 110 が全ての K レジスタを見出せると期待する記憶場所に、全ての K レジスタがあることを保証する。SYNC はレ

ジスタのアロケータに対するディレクティブ (指示) として作用し、修正されたマシンレジスタにある擬似レジスタをフラッシュすべきことを示す。また、SYNC は、K レジスタを修正する効果しか持たないコードを除去するデッド (不動作) コードから、コンパイラ 104 を保護する如何なるライブ (生) データも使用する。

【 0160 】 EXTMOD 演算コードは、擬似レジスタが修正されたが、コンパイラ 104 はその修正の詳細は持っていないことを示す。従って、EXTMOD は 2 つの効果を有する。その第 1 は、定数畳込み、又はコピー伝播等の最適化に対する障壁として作用する。第 2 には、擬似レジスタの次の使用前に、コンパイラ 104 のレジスタアロケータにフィル挿入を実行させる。OOC に於いて、EXTMOD 命令はインタプリタ 110 にコールバックした後、どの K レジスタが修正されたかを指示するのに用いられる。

【 0161 】 IL データ構造

IL がどの様にして K 演算コードから構成されるかを説明する前に、コンパイラ 104 で使用される主なデータ構造に親しんでおくのは有効である。

【 0162 】 ZONE (compiler / zone. [h, c])

コンパイラ 104 に於けるメモリの割付は、ZONE と称する抽象化 (論理化) によって取り扱われる。ZONE 抽象化は効率的なメモリの割付方法であって、一挙に割付解除もできる。ZONE 抽象化によって割付は高速化すると共に、ZONE の破壊は用いた全てのメモリを再利用するので、プログラマはメモリのリークを心配する必要がない。

【 0163 】 コンパイラ 104 に於いて、ZONE は生成され、メモリを割り付ける全てのコール (即ち、通常 malloc calls) は、初期に生成された ZONE を用いて ZONE A

LLOCをコールする。コンパイラ104が実行されると、コンパイラは、全ZONEを割り当て解除するZONE_Destroyをコールする（即ち、全てのメモリの割当解除と同等のことをする）。

【0164】ZONEの基本的実施では、メモリの“チャンク（chunk：大きな塊）”を使用する。例えばZONEを生成するとき、サイズが0x2000バイトのブロックをマロック(malloc)する。ZONE_Allocに対するコールは、メモリのその“チャンク”を使い切るまで使用する。最初の0x2000バイトでは、ZONE_Allocの要求に応える余裕がない場合には、新たな“チャンク”が生成される。そして、ZONE_Allocはその“チャンク”を使い切るまで使用する。

【0165】コンパイラ104の場合、メモリは全て事前に割り付けられているため、事態は些か複雑になり、そのためマロックをコールすることが不可能である。其処で、代わりとして特別なZONEアロケータユニット（即ち、ZALLOCユニット）を使用する。ZONEアロケータは大きなメモリプール（例えば、0x10000 バイト）によって初期化される。この初期化によって、メモリはZONEが割付に使用する同サイズのチャンクに分割され、自由なチャンクのリストが保持される。従って、“マロック”要求は、メモリの自由な“チャンク”を返すZALLOC_get_chunkに対するコールによって置き換えられる。同様に、ZONE_Destroyの“free”へのコールは、ZALLOC_free_chunkへのコールによって置き換えられる。現在の実施形態では、ZONE_Allocが扱う最大割付サイズは、初期チャンクサイズである。この制限は、単に1つのサイズを扱う代わりに、可変サイズの割付を扱うようにZALLOCユニットを変更することによって、“固定化”することができる（この種のアロケータの例については、セグメントアロケーションユニットを参照）。ここで上記のような変更をしなかったのには、2つの理由がある。その第1は、可変サイズアロケータは更に複雑であって、記憶領域の断片化等の問題を生成するからである。第2は、チャックサイズは不利な影響なしに大きくできるものではないからである。チャックサイズが十分に大きい場合、もしコンパイラ104がメモリを越えて走っても、コンパイラ104はチャックサイズより大きい単一割付を要求するだけである。従って、可変サイズの割付を扱うためにZALLOCユニットを一般化することに何ら現実的利益はない。

【0166】IL_CTXT(compiler/oc_common/include/il_internal.h)

コンパイラ104は、コンパイル動作の現状トラックを保持する単一データ構造IL_CTXTを維持している。このIL_CTXTデータ構造は、現在コンパイルされているコードを表すIL_NODEの連結リストに対するポインタを記憶している。また、IL_CTXTは、ZONE及びIL_FRAME構造等、コンパイル処理を通して用いられる種々のフィール

ドを数多く記憶している。コンパイラ104の各段階は、引数としてのIL_CTXTを有し、そのデータ構造に対して、例えば多くの段階がIL_NODEを追加又は削除する等の修正を行う。

【0167】IL_NODE(compiler/oc_common/include/il_internal.h)

IL_NODEデータ構造は、K演算コードから変換されるコンパイラ104の中間言語の単一抽象命令を表す。

【0168】K演算コードから発生されるIL_NODEは、二重連結リストに保持される。このリストの最初と最後の要素に対するポインタは、IL_CTXTに保持される。このリストはコンパイラ104が現在動作しているコードを表す。コンパイラ104の各パスはこのリストを横断し、リストのコードに関する情報を発生するか、又はリストを変換する。

【0169】各LI_NODEは命令の基本性質を示す演算フィールド“op”を含んでいる。また各LI_NODEは命令のオペランドを表すオペランドフィールドのベクトルを含んでいる。オペランドフィールドの解釈は命令の演算形式に依存している。演算及びオペランドフィールドに加えて、全てのIL_NODEは、全てのノード形式によって共用される多くのフィールド、例えばノードを変換する命令のK_pc、ノードのために発生されたターゲットマシンコードの開始アドレス等を含むんでいる。

【0170】ノードに於けるオペランドフィールドの数は、演算形式によって変化する。事実、場合によっては、同じ形式の2つのノードが異なる数のオペランドを持つこともある。例えば、コール動作のためのオペランド数は、ターゲット方法に対してパスされた引数の数に依存する。このオペランド数の変化は、IL_NODEが一貫したサイズではなく、オペランドベクトルがIL_NODE構造に於ける最終項であることを意味する。オペランドベクトルは一エントリ長さであると宣言され、そしてIL_NODEは、共通フィールド及びオペランドフィールドに必要な全記憶量を計算／割り付けし、割り付けられたメモリをIL_NODEポインタにキャストすること(casting)によって割り付けされる。

【0171】全てではないが殆どの場合、各オペランドは、オペランドベクトルに関して、2つの連続したエントリを実際に要求する。オペランドは、擬似レジスタのエントリオペランド[i]に見出される。もしオペランドが変換先オペランドであれば、オペランド[i+1]は、この演算によって定義される値を用いるノードリストを指し、もしオペランドが変換元オペランドであれば、オペランド[i+1]は、値に関する定義を含むノードリストを指す。

【0172】もし演算が変換先オペランドを持っていれば、そのオペランドは常にオペランド[0]及びオペランド[1]に記憶される。

【0173】もしオペランド[i]が変換元（入力又は使

用)オペランドであれば、オペランド[i+2] もまた変換元オペランドである。即ち、全ての変換元レジスタはオペランドリストの終わりに来なければならない。

【0174】ノード中のオペランドフィールドには直接アクセスはできない。むしろ、アクセスはILOP_xxx(N)形の大きなマクロセットによって行われる。ここで、NはILNODEに対するポインタである。これらマクロは、種々の命令形式全てに関して、如何に種々のオペランドがオペランドベクトルに記憶されているかを知っている。

【0175】ノード形式のいくつかを以下に示す(このリストは、全てを含むものではない)。

【0176】単項演算

これらは、割当を含む種々の簡単な単項(1変換元オペランド)命令を表す。

形式：演算の形式

ILOP_DEST(N)

結果が向かう変換先レジスタ

ILOP_DEST_use(N)

変換先レジスタを使う命令のリスト

ILOP_SRC(N)

変換元レジスタ

ILOP_SRC_def(N)

変換元を定義する命令のリスト

【0177】2進演算

多くの2進(2変換元オペランド)命令はこのカテゴリによって表される。

形式：演算の形式

ILOP_DEST(N)

結果が向かう行く先レジスタ

ILOP_DEST_use(N)

変換先レジスタを使う命令のリスト

ILOP_SRC1(N)

第1変換元レジスタ

ILOP_SRC1_def(N)

第1変換元を定義する命令のリスト

ILOP_SRC2(N)

第2変換元レジスタ

ILOP_SRC2_def(N)

第2変換元を定義する命令のリスト

ILOP_DIVEX(N)：このオペランドは、DIV及びREM演算にのみ現れ、もしゼロによる除算例外が在ればその開始を表すノードを含む(単独)リストを指す。

【0178】LABEL

LABEL命令は、コード内のポイントを表し、このポイントに向かってブランチは分岐する。この命令は、以下のオペランドを含む。

ILOP_LABEL(N)

ラベルを識別する固有の整数

ILOP_LABEL_refs(N)

このラベルを参照する命令のリスト

ILOP_LABEL_live(N)

何れのレジスタがこのレベルで活性であることを示すBITS ET

ILOP_LABEL_rd(N)

このラベルに到達する各レジスタの定義リストのベクトル

ILOP_LABEL_misc(N)

このレベルに関する専用情報(private info)を掛ける(hang)ための何れかのパスの場所

【0179】GOTO

GOTO命令は、ラベルへの無条件ブランチを表す。

ILOP_LABEL(N)

ターゲットラベルを識別する固有の整数

ILOP_LABEL_refs(N)

このターゲットLABEL命令の単独リスト

【0180】CGOTO

CGOTO命令は、ラベルへの条件付ブランチを表し、GOTO命令と同じオペランドを含むと共に、幾つかの追加オペランドを含む。

ILOP_COND(N)

レジスタは分岐するための条件を含んでいる。このレジスタはブール形式の値を含む。この分岐は条件がTRUEであるときに行われる。

ILOP_COND_defs(N)

このレジスタを定義する命令のリスト

ILOP_COND_live(N)

分岐が行われない場合、何れのレジスタが活性であることを示すBITSET

【0181】特殊な命令マクロILOPに加えて、何れの命令にも使用できる多くの汎用マクロがある。

ILOP_HasDEST

命令が変換先レジスタを有していれば、TRUEを返す。この場合、ILOP_DEST及びILOP_DEST_useマクロは、この命令に関して使用できる。

IL_OP_START IL_OP_DONE IL_OP_NEXT

命令の変換元レジスタを介したくり返し動作に用いられる。IL_OP_STARTは第1のそうした変換元レジスタを参照して、IL_OP_INDEXに帰り、IL_OP_DONEはIL_OP_INDEXをテストして、変換元レジスタを参照したか否かを調べ、ILOP_NEXTは次の変換元レジスタに進むのに使用される。

IL_OP IL_OP_def

これらは、与えられたIL_OP_NEXTに対して特定の変換元レジスタ及びそれに関する定義リストを返す。これら5つのマクロは、一般に、以下の形のループで使用される。for(op=IL_OP_START(n); !IL_OP_DONE(n,op); op=IL_OP_NEXT(n,op)) } use IL_OP(n, IL_FRAME(compiler/oc common/include/il_frame.h, compiler/OOCT_Frame.c))。

【0182】IL_FRAME データ構造はコンパイルされた

コードが実行する文脈に関する情報を与えるのに使用される。フレームは、各擬似レジスタに関するサイズ及び記憶場所、擬似レジスタが他の擬似レジスタと如何に重なり合うか、レジスタアロケータで使用するのに何れのマシンレジスタが正当であるかを定める。更にIL_FRAME構造はコードをコンパイルする上で、Cスタックフレームが必要か否かを定める。OCTでは、Cスタックフレームは使用しない。

【0183】コンパイラ104では、IL_FRAME構造はOCT_Frame.cの機能によって初期化される。これらの機能は、Kレジスタ及びPSW場所に対応する各擬似レジスタを準備する。更にコンパイラ104の一時的擬似レジスタは、インタプリタ110の作業空間領域に対応するように設定される。また如何にKレジスタが重なり合うかについての情報も準備される。

【0184】NL_LIST(compiler/oct_common/[include, src]/nl_nodelist.h)

コンパイラ104がIL_NODEのリストを使用する多くの場所で、NL_LISTデータ構造は、これらノードリスト操作のための抽象化(論理化)を与える。例えば、上述のUseDef解析は、与えられた定義を用いるIL_NODEのリスト、及び与えられた使用に対する定義であるIL_NODEのリストを生成する。NL_LISTは簡明であって、ノードリストに関する生成、加算、削除、検索、並びに反復等の能力を与える。

【0185】K演算コードのIL変換

上述のブロックピッカ114が、何れのK演算コードをコンパイルするかを選択した後、K演算コードのILへの変換は3つの主な段階を含んでいる。第1の段階は、そのコードが基本ブロックに対して発生される順番を決める段階である。ブロックレイアウト法については先に述べた通りである。第2段階は、K演算コードの基本ブロックがブロックレイアウト法によって選択された時、その演算コードを調べて、それらが“論理演算コード”と組合せ可能か否かを決定する段階である。最後の第3段階は、K演算コード及びその引数に基づいて、IL発生手続きをコールする段階である。

【0186】Opcode Combination(compiler/oct_opcode_combine.c)

K演算コードの幾つかのシーケンスは、単一の“論理”コードとして書くことができる。例えば、2つのTR命令からなるシーケンスを用いて、32ビットレジスタ対の値を、それら命令の各半分をテストすることによってテストすることにした。これら2つのTR命令は、Kアーキテクチャでは利用できない論理32ビットのテスト演算コードを表す。IL形成手続きが2つのTR命令に関して生成するコードは、多かれ少なかれ、このパターンが認識された場合に生成されるコードに較べて能率的である。幸い、OCTはソフトウェアであるから、新たな演算コードを加え、パターンを認識する特別なユニット

を具備し、能率的なIL発生を代行させることは容易である。

【0187】与えられた演算コードに対して標準ILを発生する前に、OCT_opcode_combineルーチンがコールされる。このルーチンは、定義された全てのパターンについて、“論理”演算コードを使用することが適当かどうかを繰り返し試す。現在、2つのパターンだけが定義されているが、追加の組合せを定義することは簡単である。パターンの1つが一致すれば、その論理演算コードのIL形成手続きは、IL命令の生成に使用され、OCT_opcode_combineは通常のIL形成手続きをコールする必要がないことを示す“真”を返して来る。

【0188】IL形成手続き(compiler/oct_il_build.c)

各K演算コードに対して、特定のIL形成手続きがある。IL形成手続きは2種類の引数、命令のアドレス、及びオリジナル命令のフィールドである引数リストを採る。また、IL形成手続きは、ILを発生する間に、擬似レジスタ及びラベルのトラックを保持するのに使用される共用大域変数global_gen_stateを使用する。各IL形成手続きは、IL命令をIL_CTXT構造に加える。コンパイル時に検出される例外のチェックを取り扱う、数少ない特別な場合を除けば、ラベルの識別子(ラベルがもう1つ別のオリジナル命令のターゲットでなければ、ラベルは最適化過程で早期に削除される)は、一般に最適化を実施しようとせず、それを後段のコンパイラ104の段階に残すので、全てのIL発生ルーチンは、オリジナル命令のアドレスを有するLABEL_IL_NODEを生成する。

【0189】IL形成手続きの殆どは、コードが発生されるIL及びオリジナル命令が一旦分かれば簡単である。このコードの理解を助ける幾つかのヒントがある。

【0190】ILの形成は与えられた演算コードのコンパイル動作がデバッグに際して、容易にoffできるように設計されてきた。これは主として、REALLY_COMPILEマクロ及びCOMPILE_SECTION_Xマクロによって制御される。REALLY_COMPILEがoffされると、全てのIL形成ルーチンは、簡単にインタプリタ110へのコールバック(又はジャンプ)を形成する。COMPILE_SECTION_Xがoffされると、区画番号Xの演算コードに対する全てのIL形成ルーチンは、簡単にインタプリタ110へのコールバック(又はジャンプ)を形成する。

【0191】ILには所定の形式があるから、正しい形式を持った正しいサイズの擬似レジスタを使うことが重要である。例えば、16ビットの値を32ビットのレジスタにロードするには、先ず初めに16ビットロードを16ビットレジスタに対して実施し、次いでCVT演算を用いて16ビットの値を32ビットの値にキャストする(LOAD_CVT32マクロがこの動作を行う)。

【0192】インタプリタへのコールバック又はジャンプを挿入する際には何時でも、インタプリタ110がKレジスタに対して正しい値を持っていることを確かめるため、SYNCを追加しなければならない。コンパイルされたコードは、ESIレジスタの値をそのままに保とうとはしない(事実、それは他の値を保持するのに用いられる)。それ故、発生したコードはインタプリタ110をコール又はそれにジャンプする前に、正しい値をESIに入れなければならない。またコールバックをする場合、コードはコールバックによって修正された全ての擬似レジスタに対するEXTMOD命令を含んでいなければならない(MODIFIES REG マクロがこれを実行する)。

【0193】例外条件(例えば、オーバフロー)を扱うコードはインラインには設けられていない。その代わり、コードはIL命令リストの終わりで発生される。斯うして共通ケースをフォールスルーとしてコンパイルすることができ、発生されたコードの性能が典型的に改善される。

【0194】エントリポイント、割り込みポイントブロックピッカ114によって選択された各K演算コードに対して発生されるILに加えて、ILはエントリポイント、割り込みチェックに対しても発生される。

【0195】最適化動作をより多く起こすために、全てのブランチ変換先は外部エントリポイントとして含まれてはいない(外部エントリポイントは最適化動作に対してバリアとして作用する)。特に、外部エントリポイントの中へ作られる変換先だけが、セグメントの外側からのジャンプの対象となる変換先である。与えられたセグメントをコンパイルするとき、何れの変換先がブランチログに於るこの基準に適合するかについての部分情報は利用可能である(ブランチログに関する情報については先の説明を参照)。コンパイラ104はこの情報を使って、何れの基本ブロックが外部エントリを持っているかを選択する。これらエントリの各々に対して、ENTRY IL NODEは、エントリオリジナル命令に対して発生されたILにジャンプするGOTO IL NODEと共に発生される。

【0196】OOCTスペックは、コンパイラ104が如何なるループにも割り込みチェックを挿入すべきことを指示する。ILを発生する際、セグメント内の逆方向ブランチに対し計算されたジャンプ命令以前に、割り込みチェックを入れることによって、保守的概算を行う。割り込みチェックは、基本ブロックの最終オリジナル命令に対するラベルの後に入れられる。他の例外条件の場合、割り込みのためのILコードはラインから発生されるから、通常の場合は条件付きブランチの簡単なフォールスルーである。

【0197】コンパイラのミドルエンドに関する説明
ミドルエンドの概容
コンパイラ104の“ミドルエンド”の主な目的はコード発生段階で、更に良好なコードが発生するように、

ILの品質を改善することである。コンパイラ104の残りの部分は、ILの解析又はILを修正する変換の何れか1つを実施する一連のパスとして構成される。これらのパスの間には或る依存性があるが、それらは複数回適用できる。この点から、コンパイラの残りの部分はK命令について何ら特別な情報を持たず、ただILを処理するだけである。

【0198】この項の残りの部分を以下のように分ける。第1部では、OASSIGN 挿入を実施する段階について述べる。また、第2部では、コンパイラ104の解析パスについて述べる。最終部では、コンパイラ104の変換パス(主な最適化を実施する)について述べる。

【0199】OASSIGN 挿入(compiler/oact add_overlap_defs.c)

図15はOASSIGN 挿入の一例を示す。OASSIGN 演算コードは、擬似レジスタ間のエイリアシング(aliasing)を明確にする特別なマーカ命令である。OOCTに於いて、或るK演算コードは16ビットのレジスタを使用し、一方では、他の演算が16ビットレジスタとは別の32ビットのレジスタを使用するので、OASSIGN に対する必要性が生じてくる。OOCTでは、16ビット及び32ビットのレジスタ全てに対して別々の擬似レジスタが使用される。それ故、擬似レジスタの或るものは、互いに暗黙の重なりを形成する。その結果、2つの問題が起こってくる。まず、第1の問題は不正確な変換を実行する最適化パスの問題である。各擬似レジスタの定義に関して、コンパイラ104はその定義を使って命令のトラックを保持し、そして各擬似レジスタの使用に関しては、コンパイラ104はその定義のトラックを保持する。この情報はuse/def 情報と呼ばれる。コンパイラ104は、例えばConstant Folding/パス等のパスではこのuse/def 情報を使用する。擬似レジスタが互いに別物である場合には、use/def 計算とその情報を使うコンパイラ104のパスが要求されるため、事態は更に複雑となる。擬似レジスタの重なりから起こる第2の問題はレジスタの割当問題である。レジスタアロケータが2つの重なり合う擬似レジスタを同時にマシンレジスタに割り当てる場合、1つのレジスタに対する修正は他のレジスタを無効にすることを要求する。一般に、情報トラックの保持は非常に困難で、不必要な複雑さを惹起する。

【0200】これらの難問に取り組み、それらをコンパイラ104の複雑さに加える代わりに、コンパイラ104が問題を無視できるような、特別マーカのOASSIGN 命令の挿入方法が設計された。この方法によれば、IL発生直後、特別コンパイラはこのOASSIGN を挿入する。このコンパイラ104のパスの後、他の解析パスは擬似レジスタと重なりを持っていないと仮定することができる(use/def 解析に関して)。更に、レジスタ割り当てはOASSIGN を用いて全く容易に取り扱われる。レジスタアロケータがOASSIGN に到達すると、アロケータはその定

義に於いて変換元を除いて、OASSIGN の後に変換先を入れる。この方法は別のメモリを使用して、重なり定義の如何なる使用も、正しい値を使用していることを保証する。

【0201】OASSIGN 挿入は2つの段階で取り扱われる。その第1はUseDef解析の特別バージョンが実行される段階である。UseDefのこのバージョンは擬似レジスタの重なりに気付いて、重なり擬似レジスタを含む使用リスト及び定義リストを生成する。コンパイラ104の残りの部分は、重なり擬似レジスタを含むuse/def リストを取り扱うようには用意されていないため、Use/Def に対するこのオプションは、一般には使用されるべきではない。この解析が行われた後、手続きOCT_AddOverlap_DefsはOASSIGN を実際に挿入する。OASSIGN は重なり定義（即ち、使用擬似レジスタと重なる擬似レジスタを定める定義）を持つ全ての使用、及びラベルに於ける重なり到達定義に関して挿入される。

【0202】図15はOASSIGN が挿入される場合の例を示す。この例では、擬似レジスタGRPAIR1とGR1が重なり合っているので、コードの第1ラインのGRPAIR1 に対する割付は、GR1 の暗黙の修正である。OASSIGN はこれを明確にする。

【0203】解析パス

UseDef(compiler/oc common/src/oc _ usedef.c)
与えられた定義の使用、及び与えられた使用に対する潜在的定義の計算は、最も基本的なコンパイラ104解析の1つである。全てのコンパイラ104の最適化パスはuse/def 情報を使用する。各IL命令は(a dest)に書き込まれる1つの擬似レジスタ引数、及び(a src) から読み込まれる1つ又はそれ以上の擬似レジスタ引数を有している。Use/Def 解析の後、各destはその解析に関連し、その値(du chainと呼ぶ)を使用する全IL命令に対するポインタを記憶するリストを有する。同様に、各srcはその解析に関連し、その値(ud chainと呼ぶ)を定義する全IL命令を記憶するリストを有する。use/def 情報を計算する方法を以下に述べる。この方法は、固定点への到達を試みる反復方法（即ち、繰り返しても変化が起これなくなるまで繰り返す方法）である。

【0204】如何なるラベルに於いても、到達定義に変化が起これなくなるまで、以下のステップを繰り返す。regdefs（擬似レジスタによって指標付けしたNL_LISTs 配列）に於ける各擬似レジスタに関する定義リストをクリアする。静的プログラム順にIL NODEs を繰り返す。もし命令が擬似レジスタを使用していれば、擬似レジスタの定義をregdefs からオペランドのud chainにコピーする。もし命令がブランチであればregdefs をブランチのLABEL に記憶されている到達定義と組み合わせる。到達定義の変更によって、全ループが繰り返される。もし命令がLABEL であれば、regdefs を既にLABEL に在る到達定義と組み合わせる。もし命令が擬似レジスタ

を定義していれば、regdefs の定義リストがこの命令だけを含むようにセットする。もし命令が無条件ブランチであれば、regdefs 配列を次のLABEL に記憶されている到達定義のセットに変える。これは、次の理由から実行される。即ち、命令はそれらの静的順に処理され、そして無条件ブランチへの到達定義は、その静的サクセサに到達する定義とは同じではないからである。

【0205】ライブ変数解析(compiler/oc common/src/oc _ usedef.c)

もう1つの解析の形は、ライブ変数情報に対するものである。ライブ変数解析は主にレジスタの割付に用いられるが、誘導変数変換及びデッドコード削除にも使用できる。擬似レジスタは、もしそれが再定義される以前に実行経路に沿って用いられる場合には、プログラムの特定の点に於いてライブ変数と考えられる。またライブ変数解析は与えられた擬似レジスタの最終使用をマークする（擬似レジスタの再定義以前に、擬似レジスタが使用される可能性のある実行経路がなければ、その使用が最終使用となる）。ライブ変数情報の計算に用いる基本的な方法を以下に述べる。この方法は固定点に到達するまで、コードに関する逆方向パスを繰り返し実行する。

【0206】如何なるラベルに於いても、到達定義に変化が起これなくなるまで、以下のステップを繰り返す。ライブ変数をクリアする（擬似レジスタのbitset）。逆静的プログラム順にIL NODEs を繰り返す。もし命令が擬似レジスタを使用していてかつそれを最終使用としてマークする以前にライブでなければ、擬似レジスタのビットをライブ変数にセットする。もし命令がブランチであれば、ライブ変数をブランチのLABEL に記憶されているライブレジスタと組み合わせる。ライブレジスタの変更によって、全ループが繰り返される。もし命令がLABEL であれば、ライブ変数を既にLABEL に在るライブ擬似レジスタと組み合わせる。もし命令が擬似レジスタを定義していれば、擬似レジスタをライブ状態からクリアする。もし命令が無条件ブランチであれば、ライブ状態をクリアする。これは、次の理由から実行される。すなわち、命令をそれらの逆静的順で処理するため、無条件ブランチに於けるライブ変数はそのサクセサのものとは同じではないからである。

【0207】レジスタの割付(compiler/oc common/src/oc _ regalloc.c)

コンパイラ104に於けるレジスタの割付は二段階で実施される。第1段階ではコードの解析を行って、ターゲットマシンの高レベルモデルに基づいて一組の推奨レジスタ割付セットを決定する。第2段階では、物理レジスタを使用するために、第1段階に於ける解析結果を更に抽象性の少ないマシンモデルと共に用いて、実際にコードを修正する。この項では、第1段階について説明する。

【0208】レジスタ割付方法はグラフ着色を用いる伝

統的技術に基づいて行われる。“グラフ”のノードは、重なり合うライブ範囲の間にあるエッジを持つ擬似レジスタのライブ範囲である。N個のカラーによるグラフ着色は、各ノードに対してN個のカラーの1つを割り当てる。従って2つの連結したノードが同じカラーを持つことはない。もしライブ範囲のグラフがN個のカラーで着色できれば(N:利用可能な物理レジスタの数)、レジスタは各ライブ範囲に割り付けられる。しかし、不幸なことに、このグラフ着色にはNP困難(NP-hard)問題(即ち、それは指数的時間を要求する)があるため、実際には、発見的(試行錯誤的)方法が用いられる。

【0209】レジスタ割付は複雑な多重ステップ処理で構成されている。これらのステップを以下に詳しく説明する。

【0210】1. 独立ライブ範囲の分割、及びREGINFO構造の割付

これらの処理は、ComputeRegInfo機能によって行われる。この機能は各擬似レジスタを独立なライブ範囲に分割し、その各々に対してREGINFO構造を割り当てる。REGINFO構造はレジスタ割付に用いられる問題のライブ範囲に関する情報を保持するのに使用され、最終的には“ターゲット”レジスタ(ライブ範囲に割り当てられる物理レジスタ)を保持する。擬似レジスタライブ範囲(論理構造)とREGINFO構造との間には1対1の対応があるので、REGINFO項はライブ範囲とデータ構造の両者を参照するのに屢々使用される。

【0211】ComputeRegInfoは、REGINFO構造を割り付ける際の殆ど副次的効果としてライブ範囲の分割を行う。ComputeRegInfoは、未だREGINFOを持たない定義から始めて、それに対して新たなREGINFOを生成し、次いで繰り返しその使用の全て及びそれらの定義の全て(及びそれらの使用の全て...)を調べて、新たなREGINFOを到達可能な全ての定義及び使用と組み合わせる。

【0212】一旦、全てのREGINFOが生成されると、それらは“簡単な”ものと、“複雑な”ものとに分けられる。“簡単な”REGINFOは、正確に1つの定義と1つの使用を有している。使用は定義の直ぐ後に続く。使用はBINOP(ターゲット特別要件)の第2オペランドではない。

【0213】その他のREGINFOは全て複雑である。各REGINFOには固有のIDが与えられる。複雑なREGINFOは、[0..c->ri_complex)の範囲にあり、簡単なものは、[c->ri_complex..c->ri_total)の範囲にある。この分割の目的は、全てのREGINFOにBITSETとして記憶されているコンフリクト(競合)マトリックスを保持するメモリを節約することである。

【0214】2. コンフリクトと互換性の計算
次のステップは、REGINFO構造のコンフリクトグラフを計算するステップである。2つのREGINFOは、それらのライブ範囲が重なり合うとコンフリクトを起こす。しか

し、もしそれらがコピーで接続されていれば互換性を持つ。コンフリクトするREGINFOは同じレジスタには割り当てられない。何故なら、それらは同時に生きているからである。2つの互換可能なREGINFOは出来れば同じレジスタに割り付けすべきである。そうすればコピーは削除される。

【0215】コンフリクトはグラフ(各REGINFOに対するノード、及び各REGINFOノードとそれとコンフリクトする各その他のノードを結ぶ無向性エッジ—これはグラフ着色法によって用いられる視点である)として、又は対称2進マトリックスとして考えられる。後者の形はコンフリクトが実際に記憶される仕方により近い。

【0216】各REGINFOはコンフリクトマトリックスの一行(部分)である単一BITSETを含んでいる。2つの簡単なREGINFOはコンフリクトしないから、マトリックスの下部右4分画は全て0である。また、マトリックスは対称であるから、上部右4分画は下部左の入れ換えである。結果として、マトリックスの左側は全て記憶される必要がある。従って、コンフリクトBITSETは、c->ri_totalの代わりに、それぞれc->ri_complexビットだけである。

【0217】2つのREGINFO、A及びB、がBITSETとコンフリクトしているか否かを決めるには、先ずそれらが簡単か又は複雑かを見るためのテスト(idとc->ri_complexとの比較)が必要である。もし何れかが複雑であれば、そのIDに対応するビットを、他のREGINFOのコンフリクトBITSETに於いて調べる。もし両者が複雑であれば、何れかのビットを調べる；それらは同じでなければならない。もし何れも複雑でなければ、それらはコンフリクトしない。

【0218】コンフリクトはIL(ComputeLiveによって発生される)に記憶されたライブ情報から計算される。ComputeConflictsはILコードに関して単一パスを実行し、現在点に於けるセット擬似レジスタライブから、その点に於ける複雑なREGINFOのBITSETを発生する。各複雑なREGINFOはライブセットに加えられるから、それは既にライブセットにある全てのREGINFOとコンフリクトしている、とマークされる。各簡単なREGINFOに出会うと、それは現在のライブセットとコンフリクトしている、とマークされる。

【0219】3. “レジスタ優先順位”に関するREGINFOの分類

OC_SortRIは、種々の調整可能なパラメータに基づいて、REGINFO構造に順位を付ける。ウェイト(重み付け)パラメータは相互に関連にしているから、それら全てに定数を掛けても何らの影響も持たない。

OC_RegAllocConflictWeight:これはコンフリクトグラフのグラフ着色に置くウェイトであって、このパラメータを高く設定すれば、より多くの異なるREGINFOをレジスタに置く割付にとって有利となる。その場合、それら

REGINFO が実際にどの程度の頻度で使われるかは関係ない。殆ど使用しないREGINFO は短命になる傾向があり、長命なREGINFO には、おそらく有利に働くことに注意されたい。

OC_RegAllocDefWeight : これは定義に置くウエイトであって、この値が高ければ、多くの異なる定義のIL文を有するREGINFO にとって有利となる。

OC_RegAllocUseWeight : これは使用に置くウエイトであって、OC_RegAllocDefWeight 及びOC_RegAllocUseWeight の両者は、長命でなおかつ多くのuses/defsを持つREGINFO (但し、長い時間、使用しないで、ただ“ぶら下がっている” REGINFO ではない) に有利に働く傾向がある。

OC_RegAllocResortRate : このパラメータは、良好な着色を得るのに、どの位多くの分類をするかを制御する。

OC_RegAllocConflictWeight が0の場合、このパラメータは無関係であり、0(==無限)となるべきである。この値が小さいとき(>0)は、より多くの時間が費やされ、良好な着色が得られたことを意味する。

【0220】4. レジスタの選択

一連の制約条件が一度はREGINFO に適用される。最初の幾つかの制約条件は必須条件であって、それを適用した後、もしレジスタが残っていなければ、REGINFO はレジスタに割り当てられない(ターゲット=-1)。残りの制約条件は望ましいが、必須条件ではない。もし与えられた制約条件の何れかによって、可能なレジスタセットが空になれば、その条件は飛ばす。一旦、全ての制約条件を適用したら、レジスタセットから最も低い番号のレジスタを選んで、それを使用する。

【0221】TYPE [必須] : この形式(マシンモデルからのinfo)の値を保持するレジスタを選択しなければならない。

INUSE [必須] : コンフリクトするREGINFO (又はそれと重なり合うもの)に既に割り当てられているレジスタを選択することは出来ない。

BASEREGS [必須] : フレームがある種のフレーム/スタック/ベースポイントとして確保するレジスタを使用することは出来ない。

CLOBBERED : REGINFO の寿命期間中に誰かによって修復されたレジスタを使おうとしてはならない。

DEF CONSTRAINTS : このREGINFO を定義する各ILに対するマシンモデルからのDEST制約条件に合うレジスタの使用を試みる。

USE CONSTRAINTS : このREGINFO を定義する各ILに対するマシンモデルからのSRC 制約条件に合うレジスタの使用を試みる。

COMPATIBILITY : 既にレジスタに割り当てられた互換性リスト中のもう1つ他のREGINFO と互換可能なレジスタの使用を試みる。

【0222】一旦、全てのREGINFO のレジスタに対する

割付けが完了(又は失敗)したら、互換性制約条件を介して変化するレジスタを探すREGINFO (即ち、このREGINFO の後に割れ付けられる互換可能なREGINFO であって、ある理由から同じレジスタには入れないREGINFO) に関してもう1つのパスを実行する。

【0223】変換(最適化)パス

変換パスは最適化コンパイラ104の心臓部にある。各パスはコードの意味を残すようにし、かつ生成された最終コードが高速で走行するようにして、コードの一部書き換えを試みる。変換パスの或るものは、それ自身ではコードを改良しない代わりに、他のパスにコードの改善を行わせる。それ故、変換パスは組合せによって最良の仕事をする傾向があり、単独では効果が薄い。斯うした理由から、多くのパス、例えばDead Code Elimination パスが繰り返して実行される。

【0224】Dead Code Elimination (compiler/oc_common/src/oc_usedef.c)

Dead Code Elimination パス(OC_ElimDeadCode)は、データフロー情報及び制御フロー情報の両者に基づいて、デッド(不動作)コードの全てを除去する。データフロー情報は何ら副次的効果を持たず、その結果が使用されないIL NODEを消去するのに用いられる。また制御フロー情報は、決して実行されない全てILNODE(不到達コード)を除去するのに使用される。更に或るブランチの再対象化が実施される。使用方法を以下に説明する。

【0225】変化が生じなくなるまで、以下のステップを反復する。

1. 静的プログラム順序でIL_NODEに関して繰り返す。
 - a) もし命令が不到達であれば、それを除去する。もし命令が他の何れかの命令のターゲットではないLABEL であるか、又は次の命令へのGOTO又はCGOTO であるか、又は無条件ブランチの直後に在って、LABEL ではないければ、その命令は不到達である。
 - b) もし命令が副次的効果を持たず、それ自身以外による使用がなければ、それを削除する。
 - c) もし固定ブランチ命令が、無条件ブランチへのジャンプであれば、その命令を再対象とする(例えば、a GOTO to a GOTO)。
 - d) 他の何処か(L2)へのブランチが続く次の命令への条件付きブランチをチェックする。この場合、条件を逆にし、L2を条件付きブランチの再対象とする。

【0226】図16は、Dead Code Elimination 及びAddress Check Elimination(compiler/ooct_elim_achk.c)の例を特に示す図である。このアドレスチェック消去パスは、Dataflow解析技術を用いて不要なアドレス配列チェックを消去する。このコードは偶奇数代数に関する値の推論を実行する。言い換えれば、このコードを解析することによって、擬似レジスタが与えられた点に於いて、偶数値、奇数値又は未知数値を保持しているかどうかを決定する。この解析は大域的に実行され、ブランチ

を越えて働く。このことは、この解析がループ対して働き、また他の制御フローを介しても働き、ループの単一展開を実行する場合に特に良く働くことを意味する。使用する方法を以下に説明する。この方法は、保守的固定点への到達を試みる反復方法である。値の推論は主として3つの方法によって行われる。第1の方法では、擬似レジスタが定数に割り付けられたときに値を推論できる。第2に、擬似レジスタが既知の引数による演算結果であるとき、値を推論できる。例えば、2つの偶数の和はもう1つの偶数を与える。最後に、条件付きブランチは、擬似レジスタの値について情報を与える。例えば、擬似レジスタを均等性に関してテストすれば、1つのブランチに沿って、それが偶数であることがわかり、他のブランチに沿って、それが奇数であることがわかる。

【0227】何れかのラベルに於いて、推論値に変化がなくなるまで、以下のステップを繰り返す。

1. `invals` (擬似レジスタによって指標付けしたINVAL配列)に於ける各擬似レジスタに対する定義リストをクリアする。

2. 静的プログラム順序でIL_NODEに関して繰り返す。
a) もし命令が簡単化でき、現在既知の推論値が与えられるならば、その命令をより簡単なバージョンのものと置き換える。命令の変化によって全ループが反復される。

b) 現在の命令の実行に基づいて`invals`を更新する。
i) もし命令が条件付きであって、その命令に関して値が推論できるのであれば、ターゲットLABEL及びCGOTOに記憶されている推論値を適当な推論値によって更新する。

ii) もし命令が無条件であって、擬似レジスタを定義しているなら、`invals`に於けるその擬似レジスタの値を更新する。この値は、演算がSETであるか、又は特別な場合、例えば2つの偶数の和でなければ、未知である。

c) もし命令がLABELであれば、`inval`と既にラベルにある推論値とを組み合わせる。

d) もし命令がブランチであれば、`inval`とブランチのLABELに記憶されている推論値とを組み合わせる。`inval`の変化によって全ループは反復される。

e) もし命令が条件付きブランチであれば、その条件からの何れかの値推論は`inval`と組み合わせられる。

f) もし命令が無条件ブランチであれば、`inval`配列を次のLABELに記憶されている推論値に変える。これは、それらの静的順序に従って命令を処理するために行われ、無条件ブランチに於ける推論値は、その静的サクセザに於けるものと同じではない。

【0228】図17は、Address Check Eliminationの例を特に示す図である。解析の性能を改善するため、擬似レジスタは単純にODD、EVEN、又はUNKNOWNよりは、その他の値を取ることができる。擬似レジスタは他の擬似レジスタ、又は2つの擬似レジスタの2進演算に対し

て、EQUIVALENCEとしてマークすることができる。これによって、1つの擬似レジスタに関する情報を、他の擬似レジスタに伝播することが可能になる。例えば、擬似レジスタR1と擬似レジスタR2が等価だと分かるとする。もしこの方法がR1は偶数であることを示すことができれば(例えば、ブランチテスト結果を介して)、R2もまた偶数であるに違いない。

【0229】この方法は保守的方法であって、推論値は単調に増加しなければならないことに注意されたい。言い換えれば、もしこの方法がその実行中、値がプログラムの一点に於いて、常にEVENであると決定するならば、その値が現実的にEVENである場合に違いない。この方法は、1つの繰り返し操作中に、擬似レジスタがEVENであることを示したり、また他の繰り返し操作中に、擬似レジスタがUNKNOWNであると示したりすることは決してない。この特性から方法の終了を推論することは簡単である。

【0230】ホイスティング(`compiler/oc_common/src/oc_hoist.c`)

このホイスティング(hoisting:引き上げ)は、一般にはループ不変量コードモーションと言われ、ループに関して一定である計算を、そのループの外へ移動する処理である。この処理では、コードを各ループの繰り返しに対して一回実行する代わりに、単一時間で実行するので、有効なスピードアップが得られる。

1. ILの番号を付け直す(idは整列される)。

2. 各逆方向ブランチ(即ち、潜在的ループ)に関して、物事(things)のホイストを試みる。

a) もしループに対し他のエントリがあれば、このループからホイストされるものは何もない。

b) ループ内でIL_NODEを静的順序で繰り返す。

i) ノードが以下の条件を満足すれば、それをホイストすることができる:

(a) ノードは“実際のレジスタ”を使用又は定義しない。

(b) ノードはループ内の擬似レジスタセットを使用しない。

(c) ノードは副次的効果を持たない。

ii) ホイストできる何れかの演算(op)に関して、それが定義する何れの擬似レジスタも再命名する。

iii) ループの上にIL_NODEを移動する。

iv) 全てのIL_NODEの番号を付け直す。

v) もしブランチを検出したら、ブランチのターゲットへ飛び越える(そのブランチが実行されているか否かは、判定できないから、コードをホイストできない)。

【0231】ホイスティングパスはOCTにとって必ずしも有効ではない。その主な理由は、多くのループはエントリポイントでもあるから、それらはループへの複数のエントリを持ち、ホイスティングパスによっては調べられないからである。この問題は、ループのターゲットとして

使用される新たなラベルが生成される“ラベル分割”を実行することによって解決できる。次に、ホストされた演算はオリジナルラベルと、新たに生成されたラベルとの間にリフトすることができる。これは直ちに実行される。

【0232】共通式削除 (CSE) (compiler/oc_common/src/oc_cse.c)

共通式削除(Common Subexpression Elimination)は冗長な計算の削除を目的とする技術である。コンパイラ104は大域共通式削除(CSE)方法を使用する。

【0233】その基本的方法を図18の例と共に以下に説明する。

1. 変換先(図示例のライン1)を有する各IL_NODEに関して、変更が行われている間に、以下を実施する:

i) 変換先全ての使用をペア(対)単位でチェックし、一方が他方を支配しているか(Bへ行くのに、Aを通らねば行けない時、AはBを支配していると言う)を調べ、斯うした各AB(ライン2及び4)ペアに関して、以下を実行する。

ii) AとBが“一致”しているかを調べ、もし一致していれば、次の式のペアには行かない。AとBは“共通式”である。

iii) 以下の方法で、AとBから始まるより大きい共通式を見つける。AとBが変換先を持ち、Bの変換先が独特の使用C(ライン5)を持っていれば、Aの変換先がCを支配すると共に、Cと一致するような使用D(ライン3)を持っているかをチェックする。もし持っていれば、DとCを共通式に加え、 $A=D$ 、 $B=C$ である更に大きい式を探す。

iv) 2つの共通式A(ライン2、3)及びB(ライン4、5)を持ったので、コードをBの使用がAの代わりとなるように書き換える。もしAの変換先がBによって使用以前に変えられれば、新たな擬似レジスタに対してコピーが用いられる。

【0234】図18は、特に共通式の削除(“CSE”)の例を示す。

コピー伝播(compiler/oc_common/src/oc_copyprop.c)

コピー伝播(Copy Propagation)は、割当ターゲットの使用を割当の変換元による置き換えを試みる変換である。コピー伝播はそれ自身では、コードの品質を改善しないから、割当結果が最早使用されないコードを屢々生成してしまう。コピー伝播方法を以下に説明する。

【0235】1. 各ASSIGN演算のためには:

a) もしASSIGNの変換元が単一定義を有し、その定義の唯一の使用がASSIGNであって、そのASSIGNの変換先がその定義とASSIGNの間で、修正も使用もされない場合には、その定義を修正してASSIGNの変換先に向かう定義とし、そのASSIGNを除去する。

b) ASSIGNの変換先の各使用について、ASSIGNはその使

用だけの定義かをテストすると共に、ASSIGNの変換元がASSIGNと使用との間で、ライブであると共に有効であるかをテストする。もし両テストが真であれば、変換先の使用を変換元の使用に置き換える。

【0236】図19はコピー伝播の例を特に示す。図20は定数量込みの例を特に示す。

定数量込み(Compiler/oc_common/src/oc_cfold.c)

定数量込み(constant folding)は、コンパイル時に定数に関する演算を評価する変換である。例えば、ILが2つの定数を加算する場合、定数量込みはそれらIL命令を、加算の変換先を2つの定数値に割り当てる単一SET命令によって置き換える。

【0237】定数量込みパス方法は非常に簡単である。各IL命令は順番に検査される。各算術及び論理演算(ADD、SUB、BAND、BOR等)について、その引数の全てが定数である場合には、IL演算は変換先擬似レジスタを定引数に関する演算値にセットするSET演算によって置き換えられる。

【0238】パターン整合(Compiler/oc_common/src/oc_pattern.c)

コンパイラ104は、IL命令の既知のパターンを、更に効率的バージョンのものに置き換えるパターン整合最適化パスを有している。現在は、OOCTによって発生されるILパターンと共通に整合するパターンはない。従って、パターン整合パスは実行されない。

【0239】ターゲットコードの発生

ILが発生され、そのコードの品質を完全するための変換が実施されたあと、コンパイラ104の3つの主要なパスが、コード発生のために用いられる。この点までは、IL及び変換パスはマシン独立であったが、これら3つのパスはターゲットアーキテクチャに強く依存している。

【0240】命令の畳込み(Compiler/oc_common/src/ix86_ifold.c)

OOCTのILはRISCの様なアーキテクチャであり、修正なしでは効率的なターゲットアーキテクチャへのマッピングを行わない。特に、全てのIL命令に対しターゲット命令を発するには、OOCTのILは次善のものである。ターゲットアーキテクチャはCISCアーキテクチャであるから、複数のIL命令は屢々それらを組み合わせて、単一ターゲット命令とすることができる。命令の畳込みパスは、組み合わせて単一命令にすることのできるIL命令グループを作ることによって問題を解く様に設計される。

【0241】命令畳込みパスは、予め定められた多くの異なる命令の組合せの1つを検索する動作を行う。この場合、以下の組合せが用いられる。

・定数がADD、SUB等の種々の演算に畳み込まれる組合せ。

・SETCC命令が、条件コードに基づいて設定する命令に

畳み込まれる組合せ。

- ・同じ引数を持つDIV、REM 対と一緒に畳み込まれる組合せ。

- ・ADD、SUB 及びASL 演算を組み合わせ、単一“lea”演算、或いはLOAD又はSTORE のアドレス計算にできる組合せ。

- ・16ビットのBSWAP、STORE の組み合わせが、2つの別々な8ビットの記憶に畳み込まれる組合せ。

- ・LOAD演算の結果が第2引数をとって用いられるとき、LOAD演算が種々の演算に畳み込まれる組合せ。

【0242】命令畳み込みパスは、命令を畳み込むべきか否かを簡単に決定するが、実際に畳み込みを行うのではなく、畳み込みはマシンコード発生パスに任せる。命令畳み込みパスは、2つの方法で畳み込まれる命令をマークする。その第1は、ノードの各オペランドに“fold”ビットを用いてマークすることができる。第2に、その使用の全てを他の命令に畳み込ませた命令に、IL_COMBINE フラグ、及び命令の畳み込まれ方に関する情報を与えるmmFoldフィールドを用いてマークする。レジスタアロケータ及びマシンコード発生は、正しく動作するためにこれらのフィールドを用いる。

【0243】ターゲットレジスタ割当(Compiler/oc_cmmmon/src/ix86_regalloc.c)

レジスタアロケータ(RegAlloc)が可能なREGINFOの全てに対するレジスタを選ぶと、それらの物理レジスタを擬似レジスタの代わりに使用するためには、コードを調べ、それを修正する必要がある。更にアセンブラがそれら命令のためのコードを発生できるように、実レジスタに幾つかの追加擬似レジスタを一時的に入れる必要がある。この場合、一般に、RegAllocがそれらレジスタに置いた値を保管及び回復するため、スビル及びフィルコード(spill and fill code)の挿入が必要になる。これをするためには、OC_RegUseAlloc は制約アロケータ(GetReg)を用い、スビル及びフィルコードを挿入してレジスタを再使用する。

【0244】OC_RegUseAlloc は、コードに関して単一パスを実行し、“スタット(stat)”配列した物理レジスタの状態のトラックを修正及び保持する。このスタット配列は、如何なる瞬間も、各レジスタに在る(又は在るべき)もの、及びレジスタ又はスビル場所(又は両方)が正しいか否かを記録する。OC_RegUseAlloc はその各々が現在処理されている命令に対して、特定の修正を行う一連の段階として働く。複数のIL命令が命令畳み込みパスによって一括畳み込まれた場合には、それらは単一命令として扱われる。それらの段階は以下の通りである。

【0245】1. もし命令が物理レジスタを直接使用したら、この使用後、レジスタにフィルが発生したかを確認、RegAlloc解析によって擬似レジスタに割り当てられたレジスタを使用するように命令を修正し、全てのレ

ジスタをロックして、再利用されないようにする。

2. GetReg対する前の命令のコールによって、一時的に割り当てられたレジスタを使用するように命令を修正し、これら全てのレジスタをロックする。

3. 命令が修復したレジスタを表すスタット配列の状態情報をクリヤして、必要なスビルを挿入する。変換先レジスタを、RegAllocによって割り当てられたレジスタがもし在れば、それに変える(但し、このレジスタは、必要に応じてsrcを保持できるから、ロックする必要はない)。

4. ターゲットコード発生を必要とするレジスタに、変換元を入れるようにコードを修正する。これは、レジスタに存在する必要のある変換元オペランドのために GetReg をコールすることを含む。

5. ロックされた全てのレジスタを開錠する。

6. ターゲットコードに必要な実レジスタを使用するための変換先を固定する。これはGetRegのコールを必要とする。

7. この演算の結果を表すスタット配列を完了させ、全ての使用レジスタを固定し、それらの“前の”場所を次の命令にセットする(従って、如何なるスビル/フィルも、この完了した命令の後に置かれる)。

【0246】このスタット配列は、理解する上で重要である。これは物理レジスタ(MM_NumReg以下のレジスタは全て物理レジスタである)によって指標付けしたデータ構造の配列であって、与えられた物理レジスタ状態を示す。このデータ構造は以下のフィールドを含んでいる。

1. ri: 現在、この実レジスタと関連する擬似レジスタを識別するREGINFO 構造(無関連を示す場合は0)。これはRegAllocによってこのレジスタに割り当てられた擬似レジスタか、又はGetRegによって一時的に割り当てられた擬似レジスタである。

2. alt_ri: このレジスタに在る追加擬似レジスタを識別するREGINFO 構造。これは、GetRegが擬似レジスタを物理レジスタに割り当て、RegAllocがもう1つ別の擬似レジスタを此処(ri)に入れるときに使用される。

3. flags: レジスタの状態を識別するフラグである。例えば、RegValidは、レジスタの値が有効であることを示すのに使用される。もしRegValidがセットされていない場合には、使用する前にレジスタをフィルしなければならない。可能なフラグの完全な説明については、ix86_regallocを参照。

4. before: このレジスタにスビル又はフィルを置くべき命令。

【0247】マシンコードの発生

ターゲットに対するマシンコードは2つのパスで発生される。その第1パスは、ブランチオフセットが計算できるように、命令のサイズを決めるのに用いられる。また、第2のパスは実際のコード発生を行う。2つのパス

は、第1のパスがコードをスクラッチバッファに発生し、正しいブランチオフセットを持っていないことを除けば同じであるから、全てのコードは共用される。

【0248】両パスは、順にIL命令を通る単一パスからなっている。各命令に対して、演算コードとその形式によって指標付けされたテーブルは、コード発生機能を検索するのに使用される。これらコード発生機能は、ターゲットの詳細を熟知する必要なしに、一般化されたターゲット発生法であるEMITマクロを使用する (ix86Asm . Emit. {h,c} 参照)。これらのマクロは、ターゲットアドレスモードの何れをも使用する命令のアセンブリを容易にする。

【0249】セグメント管理

OOCTによってコンパイルされるコードは、SEGMENT データ構造に記憶される。セグメント管理には、これに関連する多くの重要な問題がある。そこで、第1に、セグメント記憶を扱う特別なメモリアロケータについて説明する。第2に、如何にセグメントを発生し、システムに導入するかを説明する。第3に如何にセグメントが削除されるか (このオプションがonの場合) を説明する。そして最後に、セグメント削除がonの場合に用いられるセグメントロッキングについて説明する。

【0250】セグメントアロケータ (compiler/SegAlloc. {h,c})

OOCTに於けるセグメントに関する記憶管理は、特別なアロケータによって取り扱われる。OOCT初期化の際、セグメントアロケータ (SegAlloc) は、メモリの大きいチャンクによって初期化される。次いで、SegAllocユニットは、可変サイズメモリの不使用チャンクに対して、前に割り当てられたメモリチャンクの開放を要求すると共に、現在のメモリ使用に関する統計を要求する能力を提供する。

【0251】SegAllocは、可変サイズ割当を取り扱わねばならないため、ZONEアロケータより複雑である。SegAllocは全く標準的な割当方法を使用する。このアロケータは、チャンクの分類されたフリーリストを維持し、割り当てられたブロックに関して、32ビットのヘッダを用いてそのサイズを示す。メモリのチャンクを割り当てるため、要求サイズに合ったチャンクに関し、そのフリーリストを検索する。もしチャンクの残りが最小サイズより大きい場合には、それを分割して、その残りをフリーリストに加える。チャンクを開放するには、それをフリーリストに加える。自由化メモリのスピードは重要な因子ではないから、フリーリストは、単一フリーブロックに組み合わせられた隣接フリーブロックに関して検索される。

【0252】セグメントの生成及び導入 (compiler/ooct . trace.c, compiler/SegMgr. {h,c})

コンパイルの主要段階が完了した後、最終結果は再配置可能なターゲットコードを含むメモリのブロックであ

る。次のステップは、そのコードに対するセグメントを生成し、セグメント用に割り当てられたスペースにそのセグメントを導入することである。OOCT Install がこの機能を果たす。最初、このセグメントのための場所はZONEメモリ領域に割り当てられる。セグメントは、ブロックピッカ114によって選択された基本ブロックのリスト (それ故、セグメントが与えられたオリジナル命令を含んでいるか否かを調べるため、後で検索することができる)、及び発生されたコードによって初期化される。SEGMR Install に対するコールは、セグメントをメモリの連続ブロックに変え、そしてそれを、SegAllocユニットを使用するセグメント用に割り当てられたスペースにコピーする。

【0253】セグメントを生成し、それをセグメント割当スペースに移動した後、何れのオリジナル命令が、それらのためにコードをコンパイルさせたかを示す変換テーブルは、更新されねばならない。外部エントリである各オリジナル命令について、変換テーブルはそのエントリに対して発生されたコードの正しいアドレスによって更新される。更に、変換テーブルは、K命令が有効なエントリを有していることを示す TRANS_ENTRY_FLAGによってマークされる。

【0254】セグメント削除 (compiler/ooct . trace.c, compiler/SegDel. {h,c})

コンパイラ104が変換テーブルにエントリを書き込むとき、既に其処にあった古いエントリに上書きすることができる。インタプリタ110は古いエントリを読むことも、また古いセグメントにジャンプすることもできない。セグメントが変換テーブルへのエントリを持たず、そのセグメントを使用するインタプリタ110がない場合、そのセグメントを削除することができ、そのメモリは他のセグメントのために使用することができる。この項では、コンパイラが如何にしてセグメントの削除可能を検出し、そしてそれを削除するかを説明する。また、通信の項では、セグメントロッキング及びセグメント削除について詳細に説明する。

【0255】コンパイラ104が変換テーブルのエントリポイントを上書きする際、コンパイラは古いエントリポイントを削除リストに置く。新たなセグメントを導入後、コンパイラ104はSEGDEL_TryDeletionsをコールする。この手続きは削除リストの各エントリポイントをチェックする。もしインタプリタがエントリポイントを使用していなければ、そのエントリポイントがその後、再使用されないように削除する。

【0256】全てのセグメントはエントリポイントカウンタを有している。エントリポイントが削除されると、コンパイラ104はセグメントのエントリポイントカウンタを減算カウンタさせる。セグメントのエントリポイントカウンタが0になったとき、インタプリタ110はセグメントを使用しておらず、新たなインタプリタ11

0がそれにジャンプすることはできない。コンパイラ104はセグメントを削除し、そのメモリを解放して他のセグメントがそれを使用できるようにする。

【0257】セグメントロック

セグメントに対する各エントリポイントは、そのエントリポイントに関してロックとして作用するカウンタを有している。このカウンタは、エントリポイントを使用するインタプリタ110の数を記録する。そのカウンタ値が0より大きい間、エントリポイントとそのセグメントはロックされ、コンパイラ104はそれを削除することはできない。エントリポイントロックの最も重要な特徴は、セグメントをロック及びアンロックする命令が、セグメント自身の一部ではないことである。このことは、インタプリタ104がロックを保持しない限り、セグメントについて如何なる命令も実施できないようにしている。コンパイラ104及びインタプリタ110に関する文書は、セグメントロック機構について詳細に説明している。

【0258】他の問題

コンパイラ104については、他の項で述べるには相応しくないが、理解する上で重要な幾多の問題がある。

【0259】スタックラッピング(common/ooc_t_warp, [c,h])

コンパイラ104は最初、動的には拡大しない小さなスタックを割り当てられる。しかし、都合の悪いことに、コンパイラ104は多くの再帰的手続きを用いるので、それが必要とするスタックのサイズが、用意されたものより大きくなることが屢々起こる。GranPower に関するプログラムを実行している間に、コンパイラ104が回復させられないページフォールトが、スタックオーバーフローに起因して起こる状態が見られる。コンパイラ104のセクションを書き換えたり、又はスタックオーバーフローによるページフォールトの正しい取り扱い方を決めてみたりする代わりに、OOC_T_bufferから割り当てられたものより更に大きいスタックが用いられる。このスタックのサイズは、それが決して制限因子（ZONEサイズのような他の因子は大きな制限となる）とはならないように選択された。斯うしたスタックを使用するため、クリーンなインタフェースOOC_T_warp Stack が設計された。このインタフェースは、OOC_Tの大きなスタックスペースを用いる機能をコールすることができる。OOC_T_warp Stack から帰ったときも、スタックポインタは変化されない。従って、コンパイラ104がシードをコンパイルするために、ooc_t_compile_seedを介して主要なエントリポイントに入っても、それはOOC_T_warp Stack を使用したと見なされる。

【0260】表明(common/assert.[c,h])

コンパイラ104のコードは、沢山の表明(ASSERTION)文を有している。この表明文は一貫性の制約及びその他エラー状態をチェックするために、コンパイラ104の

至る所で使用される。表明文は2つ主要な役割をする。デバッグ環境では、表明不履行は、問題を調べて見つけるのに有効な情報を表示又は記憶する間、プログラムを停止させる。また、本番（コンパイル）環境では、表明はエラー状態をキャッチするのに使用され、そうした状態が発生した時には、コンパイル動作から安全に抜け出すのに使用される。例えば、もしコンパイラ104がメモリを使い切ったら、コンパイラ104によるそのシードコンパイルを停止させる。

【0261】サービスルーチン (common/service.h)

サービスユニットは、KOI モニタによっては提供されないprintf及びmemset等の標準Cライブラリに於いて典型的に提供されるサービスを提供する。このユニットはこれらシステムコールを、ウィンドウズ及びファームウェア構造に於いて、違った扱いをする必要性を取り除くことを意図している。これらサービスルーチンの基本的実行対象は2つ有って、1つはウィンテスト(Wintest)プロジェクトに対してであり、他の1つはファームウェア構造に対してである。

【0262】VIII. ウィンドウズのテスト環境

ウィンドウズのテスト環境は、OOC_Tシステムの高速開発及びテストに於いて、大きな役割を果たす。ウィンドウズを用いた開発によって、標準デバッグツールがMS VCのもとに用意されている。更にプロファイラ(profile r)等の有効なツールが利用可能となっている。テストを目的として、テストスピードを上げかつその適用範囲を広げる特別なテスト方法がウィンドウズを使って開発された。

【0263】この項では、まず模擬的Granpower 環境について説明し、次いで進歩したテスト技術の殆どを実行する比較ユニットについて説明し、最後にコンパイラ104のコードダンプについて説明する。

【0264】模擬的 GRANPOWER環境

OOC_Tの初期テスト、更に進歩したテスト、及び性能解析を実行するため、ウィンドウズ環境下で走行するインタプリタが必要であった。インタプリタ110自身は修正を必要としなかったが、GranPower に供給される初期化コール及びAOIシステムコールを書き込む必要があった。OOC_Tがウィンドウズ環境下で走行するためには、コンパイラ104がインタプリタ110から個別タスクとして走行するから、多重“タスク”の実行が必要とされた。

【0265】初期化

ウィンドウのもとで模擬環境を生成する第1の段階は、KOI データ構造を正しく初期化するコードを生成し、OOC_Tタスク用のKOI の初期化API を模擬化することであった。インタプリタ110は何れのコードも実行できるように、多くのデータ構造が適当に初期化されることを期待する。更に或るデータ構造の要素は、OOC_Tを使用するかどうかを制御する。我々は初期化コードの基礎をファ

ームウェアの初期化処理に置くことによって、インタプリタ110を走行させるための正しい初期化をシミュレートすると共に、その動作を制御した。同様に、KOIの初期化APIは、OOCTタスクを実行するため、ファームウェアが使用するコードに基礎を置いた。これは標準ウィンドウズのデバッグ環境下で動くインタプリタ110(例えば、OOCT Initに対するコール)間に於けるインタフェースの初期書込み及びテストを可能にする。またこれはインタフェースの変更及びテストを簡単にする。

【0266】AOIシステムコール (wintest/MiscStub s.c, wintest/MsgStubs.c)

インタプリタ110は、利用可能なAOIシステムコールの全てを持つ環境下での実行を期待する。実行可能なものをコンパイル及びリンクするためには、AOIシステムコール用スタブ(stub)を生成する必要がある。システムコールの多くは、ウィンドウズ環境下でシステムテストを実施している間は意味を持たないので、それらコールは単に空(empty)機能として(単にリンクage目的だけに)残される。AOIシステムコールはタイミング(ScGtmSet, ScGtmRef)及びメッセージ(msgAlc, ScMsgSnd, ScMsgRcv)のために用意され、実施される。

【0267】OOCTは、Execとコンパイラ104によるプロセス間通信のためのメッセージ受け渡しシステムに大きく依存する。ウィンドウズ環境下では、それらAOIシステムコールのダミーバージョンは、同じタスク内のスレッド(thread)が通信(上記の)することを可能にする。メッセージシステムコールのウィンドウズバージョンは、ロッキング及びメッセージ待ち行列を使用するシステムコールの仕様を完全実施する。

【0268】Compiler/EXEC用分離スレッド
ウィンドウズ環境下での実施及びデバッグを簡単にするため、分離処理の代りに、コンパイラ104及びインタプリタ110に対し分離スレッドを用いた。スレッドを用いることによって、2つの“タスク”間に於けるメッセージ受け渡しは簡単に実行される。またデバッグは2つの理由から更に容易になる。即ち、デバッグはそれぞれ単独で両方のタスク(インタプリタ110とコンパイラ104)に使用できること、そしてデバッグは複数スレッドに使用できるように設計されているからである(我々は多重処理をデバッグするツールを備えたデバッグを知らない)。

【0269】比較ユニット

OOCTは非常に価値があると証明された独特のテスト方法を使用する。OOCTのコンパイルされたコードは、インタプリタ110の結果と正確に同じ結果を生まなくてはならないから、それらの結果を直接比較する方法が生成された。ウィンドウズのテスト環境下では、OOCTとインタプリタ110の両者のもとでプログラムを走らせ、中間結果を最小単位で比較する能力が内蔵されている。これ

らの比較は、命令毎にチェックするように任意に細かく分けることができる。プログラムの動作を比較する能力と共に、自動テスト発生器が書き込まれている。このテスト発生器は、実行され、比較される“ランダム”コードを生成する。この自動テスト発生及び比較は、OOCTが正しく動作していることを確かめる極めて大きなプログラムの組を用意する。更に、自動比較はコンパイルされたコードとインタプリタ110が最初に異なる場所を指摘するから、発生したバグをピンポイントで指摘する極めて価値のある方法を提供する。

【0270】この項では、比較ユニットを2つの段階に分けて説明する。第1段階では、コンパイルされたコードの結果と、インタプリタ110の結果とを比較するインフラストラクチャについて説明する。第2段階では、テストで使用するランダムコードの発生について説明する。

【0271】比較インフラストラクチャ

比較インフラストラクチャは、同じKプログラムの2つのバージョン実行するという考えに基づいている。そこでは模擬Kマシン(レジスタ及びメモリ)のマシン状態を特定の回数だけ、チェックポイントで検査する。次いで、それらチェックポイントでの検査の結果を比較し、コンパイルされたバージョンと解釈された(interpreted)バージョンが同じ結果を与えているかを決定する。

【0272】図21は本発明の実施形態による比較インフラストラクチャを有する上記処理の例を特に示す。実際には、この比較テストは2つのウィンドウズ処理として実行される。親(主)処理はブランチロギング及びコンパイルを持つ完全なOOCTシステムを実行し、子(副)処理はKOIの解釈されたバージョンだけを実行する。これら両処理はそれらのチェックポイントログをメモリ(子は共用メモリ)に書き込み、模擬Kマシン状態への両処理の効果(影響)を記録する。親処理はチェックポイントログ内のデータを比較し、何れかの矛盾(不一致)が在ればそれを報告する。

【0273】コード発生

比較テスト用のランダムコード発生は3つのユニットによって行われる。まず、KアセンブラはC機能コールを用いて、Kマシンコードを作る機構を用意する。第2のユニットはK演算コードの種々の基本ブロックを生成するために設けられる。そして、最後はランダム制御フローユニットで、多くの異なる形式の制御フローを持つコードを発生させる。

【0274】Kアセンブラ (wintest/OOCT Assemble. [h,c])

KアセンブラはCプログラムからKコードを発生するための簡単な機構を用意する。各K演算コードはその演算コードに対する命令を特別にアセンブルするのに使用する機能を有する。個々の命令は引数として、コードを何処に記憶するかを指定するポインタ、ラベル(多分空)

の名称、及び命令に使用されている各フィールドに対する引数を取る。この機能はフィールドをそれらの正しい場所と組合せ、コードをバッファに書き込む。ラベルに対するブランチはラベルの定義以前に起こるから、コードに関する第2パスはブランチの変換先を分析するのに使用される。

【0275】ランダムK演算コード生成ユニット (wintest/GenArith.c, wintest/GenCassiest.c, Wintest/GenMisc.C)

種々の形式の命令をテストするため、それら形式の命令を含む基本ブロック(ストレートラインコード)を発生する各ユニットが生成される。特に、算術及びシフト演算、Cアシスト命令、及びOOCTによって実行される他の全ての命令を発生するユニットが生成される。これらユニットに対する主要インタフェースは、FillBasicBlockルーチンを介している。このルーチンは引数としてメモリバッファ及び多数の命令を取り、与えられた多数の命令(ランダムに選ばれた)をバッファに書き込む。FillBasicBlockルーチンは機能発生命令配列からランダムに選んで、命令を加える。ユニットは発生できるK演算コード毎に、1つの命令発生機能を含んでいる。この命令発生機能は、アセンブラに対する引数として適当なランダムな値を選び、命令をアセンブルする。命令は完全にランダムには発生されない。その代わり、ある制限のものに発生される。例えば、レジスタをランダムに選んで変換先とする場合、決してベースレジスタを使用しない。また、コードが前もって決められた多くのメモリ場所を使うことも制限される。我々のテストでは、これらの制限は非常に重要であるとは示されなかった。もし将来、それが重要であると分かれば、更に複雑な処理を用いてその制限の幾つかを緩和することはできる。

【0276】ランダムテストは多くの異なる命令間の相互作用をテストするので重要であり、OOCTのようなコンパイラ104にとって特に重要である。OOCTに於いて、命令をコンパイルすることによって作られるコードは、実質的に周囲の命令に依存して異なることができる。

【0277】図22は、異なる周囲命令に対して同じ命令のためのコードを発生するコード発生例を特に示す。更にランダムテストは、プログラマがテストしない多くの場合をテストする。

【0278】ランダムK演算コード生成ユニットは、ある種のテストに対してそれ自身で有効である。例えば、新たな演算コードを実施するとき、このユニットは、その演算コードを用いる命令の基本ブロックを実行する簡単なループを生成する非常に有効な方法であることを立証している。個々のユニットが有効であるとは言え、コンパイラ104の幾つの特徴を完全にテストするには、もっと複雑な制御フローが必要である。

【0279】ランダム制御フロー生成ユニット(wintest/Gdom control flow creation unit(GenControl))は、

ストレートラインコードより更に複雑な形式の制御フローを用いるテストを生成するのに使用される。GenControlは単一基本ブロックから始めて、幾つかの変換(ランダムに選んだ)を実施する。現在実施されている変換は下記の通りである。基本ブロックは2つの基本ブロックに分割できる。基本ブロックはダイヤモンドによって置き換えることができる。これは条件付きブランチを表し、其処では2つの経路が結合し直して一緒になる。基本ブロックはループによって置き換えることができる。基本ブロックは3つの基本ブロックによって置き換えることができ、其処では第2の基本ブロックに対して機能コールが行われ、第3基本ブロックに戻る。

【0280】基本ブロックに関して、特定数の変換が実行された後、命令で満たす必要があるランダム発生制御フローグラフが存在する。これは2つの部分からなっている。第1の部分では、基本ブロック自体に対するコードを発生するため、前項で説明したランダムK演算コード生成ユニットが使用される。第2の部分では、命令を満たしてブランチ及びループを実行する。ループは固定回数繰り返す所定のテンプレートを使用する。条件付きブランチに対しては、ランダムテスト命令が使用される。

【0281】コンパイラコードダンプ

デバッグ及び最適化を目的として、多くのダンピング機構がウィンドウズ環境下のOOCTで使用される。主なダンピング機構は2つある。その1つはコンパイルされたK演算コード、IL、及びターゲットコード(もし発生されていれば)を含むコードリストを、コンパイルの間にダンプすることができる。第2の形式のダンピング機構は、テストを目的として、再コンパイル及びリンクすることができるアセンブリ形にターゲットコードをダンプするものである。

【0282】幾つかの段階の後、ILコードのコピーをダンプすることによって、与えられたコンパイラ104の最適化パス効果の正確性及び有効性を検査することができる。更に、作られた最終コードを検査することによって、コンパイラ104による各K演算コードのILへの変換の善し悪し、及び各IL命令に対して作られたターゲットコード及びK演算コードの品質をマニュアルで検査することができる。これらのコードダンプは、OOCT_Optimize_IL_And_Gen_Code(compiler/ooc_t_trace.c参照)に於けるコンパイラ104のパス間に挿入されたCOMBDUMPマクロを用いて制御される。このマクロは、K演算コード及びIL命令に関して繰り返すOOCT_Combdump手続き(ooc_t_combdump.c参照)をコールする。

【0283】ウィンドウズ用の現在のプロファイリングツールは、動的に発生されたコードを正しく取り扱わない。それ故、第2の形式のダンプを用いて、1つの実行からの動的コードが、もう1つの実行からの静的コード

として使用でき、そして正しくプロファイルできるようにする。これは2段階で達成される。第1段階では、再コンパイル可能なフォーマットを持つファイルに、コンパイルされる各K演算コードのトレースを記録し、そしてコードをダンプする。第2段階では、プログラムをコンパイルして、それをOC_USEDUMP フラグ(compiler/ooct_dump.h参照)によって実行し、静的バージョンを使う代わりに前にコンパイルされたコードに対する動的コンパイルをoffする。プログラムのこのバージョンは、コードの品質に関する統計を記録するプロファイルによって実行することができる。

【0284】本発明の第2実施形態 動的最適化オブジェクトコード変換 第2実施形態の概観

アーキテクチャエミュレーションは、オリジナルアーキテクチャで使用するマシンコードを修正なしで実行できるように、1つのコンピュータアーキテクチャを他の異なるコンピュータアーキテクチャによって模倣することである。オブジェクトコード変換は1つのコンピュータアーキテクチャ用マシンコードを、他のコンピュータアーキテクチャ用マシンコードに変換する処理である。ここで述べる動的最適化オブジェクトコード変換システムはコンパイラ最適化技術を用いて、アーキテクチャエミュレーションのためのテンプレートベースのオブジェクトコード変換より高いオブジェクトコード変換性能を達成する。

【0285】第2実施形態に関する図の説明

図23は本発明の第2実施形態による動的最適化オブジェクトコード変換に用いるシステム構成を示す。図23はプログラム解釈の実行と同時発生の動的変換を示す概略図である。各インタプリタはコンパイラに対し変換要求を送ることができる。次いでコンパイラはインタプリタタスクにとって利用可能な変換されたコードを作る。多重実行ユニットを持つマシンでは、全ての処理が同時に実行される。

【0286】第2実施形態の詳細な説明

動的最適化オブジェクトコード変換システムは1つの命令セットの動的コンパイルを、もう1つ他の命令セットに対して実施し、テンプレートベースの変換又は解釈されたエミュレーションに関する性能改善を行う。動的最適化オブジェクトコード変換システムは、実行コードをプロファイルする任意の数のインタプリタを別の最適化コンパイラに組み合わせる。最適化コンパイラは実行コードからのプロファイリング情報を使用して、頻繁に実行されたコード部分を決定する。次いで、これらのコード部分はコンパイルされ、インタプリタに与えられる。このシステムの全体構造を図23に示す。

【0287】有意味のコンパイラ形式の最適化は、命令フローグラフに関する情報によってはじめて実行が可能になる。伝統的なコンパイラでは、最適化が始まる前

に、全ルーチンが完全に解析されるので、フローグラフは明瞭に定義されて与えられる。アーキテクチャエミュレーションシステムの場合、コンパイルされるコードは、それが実際に実行される前には利用可能ではない。更に命令及びデータは、実際にプログラムを走行させなければ、一般には区別することはできない。

【0288】それ故、フローグラフを決めるには、プログラムを実行しなければならない。インタプリタは最初にプログラムを走行させるのに使用される。インタプリタがプログラムを実行するとき、インタプリタはブランチ演算を行うことを動的コンパイラにその都度報告する。この情報のロギングは、幾つかの命令及び幾つかの接合点を識別する。プログラムが走行すると、フローグラフに関する情報はより完全にはなるが、全く完全とは言えない。システムはフローグラフに関する部分的情報によっても作動するように設計されている。この場合、最適化は潜在的に不完全なフローグラフに基づいて実施されるが、より多くの情報が利用可能になったとき、システムはその最適化コードを入れ替えられるように設計されている。

【0289】動的コンパイルは、インタプリタによって集められたプロファイリング情報に基づいて、テキストのどの部分を最適化すべきかを選択する。或るブランチを実行した回数が閾値を越えたとき、ブランチの変換先はコンパイルのシードになる。シードは、1単位としてコンパイルされる変換元命令の部分を解析する始点である。この単位をセグメントという。

【0290】セグメントは、シードから変換元命令を最適化した結果得られた命令を含んでいる。セグメントは1単位として導入及び非導入される。インタプリタがコンパイラをコールして、コンパイラにブランチについて知らせると、もし変換先コードが在れば、コンパイラは制御をセグメントに移すことを選択する。同様に、セグメントは制御をインタプリタに移し返すためのコードを含んでいる。

【0291】セグメントが不完全で、変換元プログラムから可能なフロー経路のサブセットしか表していない場合もある。しかし、この不完全な表現は正しいエミュレーション動作と干渉はしない。もしオリジナルコードを介して、新たに予想外のフロー経路が生じた場合には、制御フローはジャンプしてインタプリタに戻る。後で、同じセグメントは新たな制御のはがれを説明するために置き換えられる。

【0292】第2実施形態の特別目的

本発明は、アーキテクチャエミュレーションシステムに於ける性能改善のために最適化オブジェクトコード変換を使用する。

【0293】第2実施形態の要約

ここで述べた動的最適化オブジェクトコード変換システムは、アーキテクチャエミュレーションのため、コンパ

イラ最適化技術を用いて、テンプレートベースのオブジェクトコード変換より高いオブジェクトコード変換性能を達成する。本発明は、アーキテクチャエミュレーションシステムに於ける性能改善のために最適化オブジェクトコード変換を使用する。

【0294】本発明の第3実施形態 同時動的変換

第3実施形態の概容

動的変換は、1つのマシン語で書いたコンピュータプログラムを、そのプログラムの実行中に、他のマシン語で書いたものに変換する動作である。ここで述べる同時動的変換システムは、プログラム解釈の実行と同時に変換を行う。

【0295】第3実施形態に関する図の説明

図24は本発明の第3実施形態による同時動的変換に用いるシステム構成を示す。図24はプログラム解釈の実行と同時の動的変換を示す概略図である。各インタプリタはコンパイラタスクに対し変換要求を送ることができる。次いで、コンパイラタスクはインタプリタタスクにとって利用可能な変換されたコードを作る。多重実行ユニットを持つマシンでは、全ての処理が同時に実行される。

【0296】図25は、例えば1つのタスクとして実行する間、インタプリタとコンパイラを組み合わせることで、また例えば、本発明の第4実施形態に従って、異なるタスクとしてそれらを分離することとの違いを示す。図25は、インタプリタとコンパイラタスクを組み合わせた時の待ち時間と、分離した時の待ち時間の概略を示す図である。

【0297】第3実施形態の詳細な説明

同時動的変換の目的は、インタプリタが実行状態にある間に、実行プログラムを更に効率的な形にコンパイルすることによって、インタプリタの性能向上を図ることである。インタプリタの実行と同時に動的変換を行うため、コンパイラは多重実行ユニットを有するシステムに関して分離タスクとして実行される。コンパイラタスクは、或る命令を変換する要求を受け、一定の変換されたコードによってその要求に応えるサーバである。分離タスクとしてコンパイラサーバを置くことには、幾つかの利点がある。即ち、その第1は1つ以上のインタプリタタスクが、同じサーバに対して要求することができる。第2に、インタプリタタスクは、先に進む前に、コンパイル要求の結果を待つ必要がない。第3に、インタプリタ及びコンパイラは、それぞれのタスクに於ける故障から隔離される。そして、第4に、インタプリタ及びコンパイラは利用可能なプロセッサの数に応じて、仕事がより平均化するようにスケジュールを組むことができる。これらそれぞれの利点については以下の詳述する。

【0298】分離コンパイラタスクを持たない動的変換システムは幾つか現存している。サンマイクロシステム

ズ(Sun Microsystems)社のジャバ仮想マシン(Java virtual machine)はその一例である[2]。この仮想マシンは手続きを呼び出すことによって動的変換要求を発することができる。しかし、このシステムでは、インタプリタはプログラムの実行を継続する前に、変換要求が完了するのを待たなければならない。もう1つの例は、富士通社の一挙に命令ページを変換するOCT 動的変換システムである[1]。このOCT システムでは、プログラムの実行を継続する前に、変換要求が完了するのを待たなければならない。

【0299】また、ジャバの変換元コードをジャババイトコードに静的変換するのに利用できる変換サーバがある[3]。これらのサーバはジャバのプログラムが走行している間は動作しないので、動的変換ではなく、静的変換のための分離コンパイラタスクの利益を提示している。

【0300】分離コンパイラタスク構成の第1の利点は、複数のインタプリタタスクが同一サーバに対して変換要求をすることができる点にある。それらの変換要求は、それらの実行可能なイメージに、それを更に小さくするコンパイラコードを含まなくてもよく、インタプリタ命令とコンパイラ命令との間、或いはインタプリタデータデータとコンパイラデータとの間にキャッシュ競合を起こさない。効率的キャッシュの使用は、殆ど全てのモダンなプロセッサにとって重要であるから、これは大きな利点である。

【0301】分離コンパイラタスク構成の第2の利点は、インタプリタがコンパイラの待ち時間を見なくて済む点である。図25は待ち時間の差を示している。インタプリタとコンパイラの組合せタスクでは、インタプリタはコンパイラが命令変換を完了するまでは、命令を実行しない。分離タスクの場合は、インタプリタは、コンパイラが動作している間に、直ちに命令実行を再開する。分離タスクによって為される全仕事量は、変換要求をしてその回答を受けるから大きくなるが、待ち時間が小さいと言うことは、コンパイラが働いている間、システムユーザが休止時間を持たなくて済むことを意味する。また、コンパイラが働いている間も、インタプリタタスクは割り込み等の外部事象に応答することができる。これは組合せタスク構成では不可能である。実際、組合せ構成に於いて、インタプリタがコンパイラの待ち時間を経験するという事実は、コンパイラの複雑さ及び変換されたコードの品質に制限を置くことになる。例えば、ジャバのJust-In-Timeコンパイラはジャバシステムと対話しているユーザが休止を経験しないような十分な高速でしなければならない。これは複雑な最適化を禁止することになる。同様に、OCT システムはコンパイル時間を減少させるため、1つの変換された命令を最適化するだけである。分離コンパイラタスクは、複数の命令にわたって最適化の実施を可能にする。

【0302】分離コンパイラタスクの第3の利点は、インタプリタタスクとコンパイラタスクに於ける故障が互いに隔離される点にある。このことは、もしコンパイラがアドレス例外や、その他の例外状態に遭遇しても、インタプリタタスクは影響を受けないことを意味する。コンパイラは故障の後、それ自身をリセットし、次の変換要求に関する仕事を続ける。インタプリタタスクは、変換要求を終了するのにコンパイラを待つことはないから、コンパイラの故障を知らずに済む。

【0303】分離コンパイラタスクの第4の利点は、コンパイラタスクとインタプリタタスクに対する負荷のバランスを取ることができる点である。動的変換システムでは、インタプリタタスクが非常に忙しく、コンピュータのCPUの全てを必要とする時期があり、またインタプリタタスクが遊休状態で、CPUが使用されていない時期もある。インタプリタとコンパイラの組合せ構成では、コンパイラはインタプリタが実行時にコールされるだけだから、コンパイルの仕事の殆どはインタプリタの実行時に行われる。従って、これは遊休CPUサイクルの利点を使用しない。分離コンパイラタスク構成では、コンパイラはインタプリタが遊休状態にあるときでも動作を継続する。コンパイラはインタプリタが将来使うかもしれない変換されたコードを作成する。

【0304】第3実施形態の特別目的

本発明の第3実施形態は、より小さな実行可能なイメージサイズを与え、キャッシュ競合を低減し、インタプリタ実行の待ち時間を短縮し、故障を隔離し、更に良い負荷のバランスをとる複数の物理的実行ユニット有するシステムに於いて、複数インタプリタの実行と同時に動的変換を行うことを目的とする。

【0305】第3実施形態の要約

ここで述べた動的変換システムはプログラム解釈の実行と同時の変換を行う。このシステムはインタプリタタスクの実行に重大な影響を与えないように、分離コンパイラを使用する。本発明は、より小さな実行可能なイメージサイズを与え、キャッシュ競合を低減し、インタプリタ実行の待ち時間を短縮し、故障を隔離し、更に良い負荷のバランスをとる複数の物理的実行ユニット有するシステムに於いて、複数インタプリタの実行と同時に動的変換を用いる。

【0306】本発明の第4実施形態

エミュレータに関するプロファイリングの負担を低減する動的変換実行中のエミュレーション

第4実施形態の概容

アーキテクチャエミュレーションは、オリジナルアーキテクチャ用マシンコードが修正なしで実行できるように、1つのコンピュータアーキテクチャを他の異なるコンピュータアーキテクチャによって正確に模倣することである。オブジェクトコード変換は、1つのコンピュータアーキテクチャ用マシンコードを異なるコンピュータ

アーキテクチャ用マシンコードに変換する処理である。ここで述べる動的最適化オブジェクトコード変換システムは、コンパイラ最適化技術を用いて、アーキテクチャエミュレーションのためのテンプレートベースのオブジェクトコード変換より高いオブジェクトコード変換性能を達成する。しかし、動的最適化オブジェクトコード変換を実現するにはプロファイリングが必要である。ここでは、プロファイリング負担を低減する方法について説明する。

【0307】第4実施形態に関する図の説明

図26は本発明の第4実施形態に従って、どの命令が変換可能であり、どの命令が変換不能であることを記録するのに使用する変換テーブルを示す。図26はどのプログラムが変換可能であり、どのプログラムが変換不能であることを示す変換テーブルである。この場合、プログラムは1バイトの単位で測られる。エミュレータは、どのエントリにブランチサクセサが対応するかをチェックし、変換可能なプログラムにジャンプするか否かを決定する。

【0308】図27は、本発明の第4実施形態による方法が、エミュレータに関するプロファイリング負担を如何に低減するかを示す。図27は、エミュレータがどの様にして変換可能なプログラムに関してロギングをonし、変換不能プログラムに関してそれをoffするかを示す流れ図である。トリガ*1 命令及びトリガ*2 命令は、両者ともログされねばならないが、トリガ*1 命令は変換可能プログラムと変換不能プログラムの間にジャンプしなくても良い。トリガ*2 だけがそれらの間にジャンプすることができる。ログフラグはエミュレータが変換可能プログラムで走行しているか又は変換不能プログラムで走行しているかを覚えていく。従って、トリガ*1 命令では、エミュレータは変換テーブルをチェックしたり、又はログフラグを変えたりしなくても良い。従って、エミュレータはブランチサクセサ命令が既にコンパイルされているかどうかだけをチェックし、直ちにコンパイルされたコードにジャンプする。トリガ命令*1 は、最も多く実行された命令を表すから、このアルゴリズムはエミュレーションに関するプロファイリングの負担を低減する。

【0309】第4実施形態の詳細な説明

動的最適化オブジェクトコード変換は、更に高速な命令を作ることによって高い性能を実現できるが、それにはメモリ及び時間の点から見てコストが伴う。それ故、アーキテクチャエミュレーションでは、動的オブジェクトコード変換とエミュレーションの両方を一緒に用いる。この変換は最も多く実行され、高性能を必要とするメジャープログラムに対して用いられる。そして、エミュレータは変換プログラムがコンパイルを完了するまで、マイナープログラム及びメジャープログラムのプロファイリングのために作動する。変換プログラムはプロファイ

ルを用いてプログラムのコンパイル及び最適化を実施する。

【0310】未変換コードから変換されたコードにジャンプする命令をトリガ命令と言う。もしトリガ命令がマイナープログラムからメジャープログラムへ、又はその反対にジャンプする場合、その命令をトリガ*2 命令と言う。もしトリガ命令がマイナープログラム内、又はメジャープログラム内だけでジャンプする場合、その命令をトリガ*1 命令と言う。変換プログラムはマイナープログラムでは動作しないから、マイナープログラムに於いてトリガ*1 命令をプロファイルする必要はない。メジャープログラムの一部分は、他の部分がまだ変換されていない間に変換されるから、メジャープログラムに於いてトリガ*1 命令をプロファイルする必要がある。トリガ*2 命令はメジャープログラムへジャンプするから、マイナー及びメジャープログラムの両方でプロファイルする必要がある。

【0311】エミュレーションは、トリガ*2 命令を実行した後に、3つの命令を実行する(図27参照)。先ず、変換プログラムがonしているかをチェックする。もしonしていれば、トリガ*2 命令のサクセサが変換可能か否かをチェックする。もしサクセサが変換可能であれば、エミュレーションはロギングフラグを真にセットし、そしてサクセサが変換されたかをチェックし、もし変換されたバージョンが在れば、それにジャンプする。

【0312】エミュレーションは、トリガ*1 命令を実行した後に、2つの命令を実行する(図27参照)。先ず、ロギングフラグがonか、offかをチェックする。フラグがoffなら、この命令はマイナープログラムにあり、プロファイルする必要はない。フラグがonならば、エミュレーションはサクセサが変換されたか否かをチェックする。

【0313】メジャー及びマイナー両プログラムは、それらのメモリアドレスによって区別される(図26参照)。エミュレータは変換テーブルを用いて、変換可能及び変換不能プログラムアドレスの関係を記録する。変換可能プログラムと変換不能プログラムの間を決して移動しないトリガ*1 命令に関しては、ロギングフラグが既にその情報を持っているから、エミュレータは変換テーブルにアクセスしなくても良い。

【0314】トリガ*1 命令及びトリガ*2 命令に対するエミュレータの動作を分離することによって、エミュレーションに関するプロファイリング負担は低減される。

【0315】第4実施形態の特別目的

本発明の第4実施形態はエミュレータに関するプロファイリング負担を低減する方法を目指している。この方法は、変換可能な命令へ、又はそれからジャンプすることができ、ブランチサクセサが変換可能か否かをチェック

するコードをトリガ命令の後に置き、そして単にフラグをチェックして、ブランチサクセサが変換可能か否かを調べるコードを他の全てのトリガ命令の後に置くことによって構成される。

【0316】第4実施形態の要約

動的オブジェクトコードをエミュレーションと一緒に用いることは効果的ではあるが、変換プログラムを案内するプロファイリング命令のコストはエミュレーションの負担となる。形式の異なるプロファイリング命令を区別することによって、この負担を低減することが可能である。本発明は、エミュレータに関するプロファイリング負担を低減する方法であって、この方法は、変換可能な命令へ、又はそれからジャンプすることができ、ブランチサクセサが変換可能か否かをチェックするコードをトリガ命令の後に置き、そして単にフラグをチェックして、ブランチサクセサが変換可能か否かを調べるコードを他の全てのトリガ命令の後に置くことによって構成される。

【0317】本発明の第5実施形態

動的変換のためのソフトウェアフィードバック

第5実施形態の概容

動的変換は、1つのマシン語で書いたコンピュータプログラムを、そのプログラムの実行中に、他のマシン語で書いたものに変換する動作である。ある種の動的変換システムでは、プログラムを実行するインタプリタと呼ばれるタスクと、コンパイラと呼ばれるプログラムを変換するタスクとが、相互に分離される。インタプリタがコンパイラに対して要求を送る割合は、コンパイラがその要求を完成する割合に一致しなければならない。また、インタプリタが要求を送る割合がゼロに落ちてはならない。ソフトウェアフィードバックは、これら2つの割合を同じにする方法を提供する。

【0318】第5実施形態に関する図の説明

図28は本発明の第5実施形態によりインタプリタとコンパイラを分離した動的変換システムの全体構造を示す。図28は動的変換システムの構造図である。インタプリタはコンパイラに対して要求を送る。これに回答して、コンパイラは変換されたコードをインタプリタに送り返す。このシステムが最も効率的な動作をするには、これら要求と応答の割合が等しくなければならない。

【0319】図29は本発明の第5実施形態によるソフトウェアフィードバック機構の構成要素を示す。図29はソフトウェアフィードバックシステムの構成要素を示す図である。比較手続きは完成数から要求数を減算する。要求割合手続きはその差に基づいて割合を設定する。要求送付手続きは、現在の割合によって要求を送る。

【0320】第5実施形態の詳細な説明

動的変換システムでは、インタプリタタスクはコンパイラタスクに要求を送る。この要求には、プログラムのど

の区画を変換するかをコンパイラに告げる情報が含まれている。要求を何時送るかを定める問題は、スケジューリング問題の一例である。インタプリタタスクが要求する割合は、コンパイラタスクがその要求を完了する割合に一致しなければならない。それ故、コンパイラは遊休状態になったり、或いは要求過多状態になったりはしない。

【0321】ソフトウェアフィードバックは、そうした2つの事象の割合を等しくする方法である[1]。動的変換システムでは、この方法は変換要求割合を変えて、変換完了割合に等しくする。図29に示すように、ソフトウェアフィードバックシステムは3つの主要部分を有している。その第1は、変換要求数と変換完了数とを比較する手続きである。第2は、この比較に基づいて変換要求の割合を変更する手続きである。第3は、変換要求が第2の手続きの出力に依存するようにする手続きである。

【0322】動的変換システムに於いて、インタプリタタスクは、どの様な頻度でブランチ命令が特定の変換先アドレスにジャンプしたかをカウントする。このカウントが閾値を過ぎると、インタプリタはその変換先アドレスを含む変換要求を送る。この閾値はソフトウェアフィードバックによって設定される重要なパラメータである。閾値が殆どの実行カウントより低い場合には、変換要求割合は高くなる。閾値が殆どの実行カウントより高い場合には、変換要求割合は低くなる。実行カウントの典型的サイズは解釈されるプログラムによって変化するから、インタプリタの動作に自動的に適応するソフトウェアフィードバックは、閾値設定には理想的な方法である。

【0323】動的変換システムにおいて、ソフトウェアフィードバックシステムの比較手続きは、非常に簡単である。それは、コンパイラに対して送られる変換要求数と変換完了数との差を計算する。

【0324】要求割合手続きは、比較手続きによって計算された差に基づいて閾値を変える。もしその差がゼロであれば、閾値が高すぎるので、インタプリタによる変換要求送付は阻止される。その場合、要求割合手続きは閾値から一定値を減算する。もしその差がその最大許容値であれば、閾値が低すぎるので、インタプリタによる過剰な変換要求送付が行われる。その場合、要求割合手続きは閾値に一定値を加算する。

【0325】要求送付手続きは、インタプリタがブランチ命令を実行する際にコールされる。もしブランチ命令が同じ変換先アドレスに、閾値以上にジャンプした場合には、インタプリタは変換先アドレスを含む変換要求を送る。

【0326】第5実施形態の特別目的
本発明は、インタプリタタスクとコンパイラタスクとが分離された動的変換システムに於いてソフトウェアフィ

ードバック機構を用いて、コンパイラが遊休状態に入らないようにしながら、インタプリタによる変換要求送付割合とコンパイラによる変換完了割合とを等しくする。最小閾値の使用によって、コンパイラは停止する。

【0327】第5実施形態の要約

インタプリタタスクとコンパイラタスクとが分離された動的変換システムに於いて、インタプリタがコンパイラに対して要求を送る割合は、コンパイラがその要求を完成する割合に一致しなければならない。また、インタプリタが要求を送る割合がゼロに落ちてはならない。本発明は、インタプリタタスクとコンパイラタスクとが分離された動的変換システムに於いてソフトウェアフィードバック機構を用いて、コンパイラが遊休状態に入らないようにして、インタプリタによる変換要求送付割合とコンパイラによる変換完了割合とを等しくする。

【0328】本発明の第6実施形態

動的変換のためのキューイング（待ち行列作成）要求
第6実施形態の概容

動的変換は、1つのマシン語で書いたコンピュータプログラムを、そのプログラムの実行中に、他のマシン語で書いたものに交換する動作である。交換されるプログラムの各片に関して、システムは動的変換プログラムに対して要求をする。動的変換プログラムの使用中に行われる要求は待ち行列に入れられ、交換プログラムが遊休状態になったとき、それに引き渡される。待ち行列の実施は、その低減を計るため、システムコールと共用メモリ通信とを組み合わせる実施する。

【0329】第6実施形態に関する図の説明

図30は本発明の第6実施形態に従って、交換タスク作動中に、如何に待ち行列を用いて交換要求を保持するかを示す。図31は本発明の第6実施形態に従って、如何にOOCT要求の待ち行列が、安価で済む共用メモリ要求と、システムコール要求とを組み合わせるかを示す。

【0330】第6実施形態の詳細な説明

要求待ち行列の基本機能は、図30に示すように、動的変換プログラムが作動中に為された要求を覚えておくことである。何れの変換システムに於いても、同時に発生することが可能な交換の数には制限がある。典型的には、この制限は1回につき1交換と言うものである。しかし、出される要求の総数又は要求の割合には制限がない。それ故、既に交換プログラムが作動している間に、1つの交換要求が行われる可能性がある。要求待ち行列を用いれば、交換要求は待ち行列に入れられ、その要求を繰り返す必要がなくなる。交換プログラムは待ち行列からその要求を取って、交換を実行する。

【0331】OOCTに於いて、動的変換システムは複数のタスク持っている。即ち、1つは要求を取り扱う動的変換タスクであり、他の1つは、交換要求をする実行タスクである。OOCTの待ち行列作成実施は、図31に示すように、安価で済む共用メモリと、システムコールメッセ

ージとを一緒に用いて要求待ち行列を形成することによって、単純な待ち行列に改良を加える。未処理の要求がない場合、シードを実行タスクから変換タスクに伝え、変換タスクを遊休状態又はブロック状態にするには、システムコール単独で十分である。しかし、システムコールは高価につく。共用メモリを使って、要求メッセージを実行タスクから変換タスクに伝えることはできるが、変換タスクはそれらのメッセージをブロックすることはできない。それ故、変換タスクは、簡単な共用メモリ待ち行列からメッセージを連続的に受け取るように動作しなければならない。

【0332】OOCTの実施に当たっては、システムコール及び共用メモリの各機構の最良の特徴を利用する。即ち、変換タスクが既に作動している場合、OOCTは変換タスクがシステムコールメッセージを待つのをブロックできるが、共用メモリを介して要求を伝えるようにする。

【0333】図31に示すように、OOCTの要求待ち行列は実行タスクと変換タスクとの間で、2種類のメッセージを使用すると共に、両タスクによってアクセスされる共用メモリバッファを加える。最初のメッセージは、変換タスクから実行タスクに送られる。このメッセージは、実行タスクに、次の要求を送るにはシステムコールを使うよう告げる。このメッセージは、実行タスクに、変換タスクが共用メモリバッファを空にし、ブロックしようとしていることを報告する。従って、実行タスクはシステムコールを使って要求を送る。変換タスクはメッセージを受け取り、変換を開始する。システムコールを使って1つの要求を送った後、実行タスクは変換タスクが作動中であることを知ると、その後の要求を直に共用メモリバッファに送る。これはシステムコールを使用するより、遥かに安価で済む。変換タスクは1つの要求を片づけると、共用メモリバッファを調べる。もしバッファに要求があれば、その要求は取り出されて、変換される。共用メモリバッファが空の場合には、変換タスクは実行タスクに再度、システムコールを使用するよう告げる。

【0334】OOCTの要求待ち行列の利点は、実行タスクの要求送付割合が高いとき、実行タスクは共用メモリを使うことができる点と、また、変換タスクに対し要求がゆっくりにした割合では入って来るときは、変換タスクはブロックすることができる点である。

【0335】第6実施形態の特別目的

以下のクレームは日本における富士通の特許の翻訳に1節を加えたものである。すなわち、本発明は、頻繁に分岐される命令の変換を始めている間も、変換タスクにメッセージを送ることによって解釈を続行し、変換処理が既に進行している時には、変換タスクに対するメッセージを待ち行列に入れ、システムコール及び共用メモリの両機構を用いて、変換要求メッセージの送付性能を改善する方法である。

【0336】第6実施形態の要約

ここで述べた変換要求の待ち行列は、変換要求を集める一方で、他の変換を実行する機構である。この待ち行列によって、実行タスクは1つの要求を送った後、直ちにその動作を継続することができる。共用メモリとシステムコールを併せて使用することによって、変換待ち行列の効率を改善することができる。本発明は、頻繁に分岐される命令の変換を始めている間も、変換タスクにメッセージを送ることによって解釈を続行し、変換処理が既に進行している時には、変換タスクに対するメッセージを待ち行列に入れ、システムコール及び共用メモリの両機構を用いて、変換要求メッセージの送付性能を改善する方法である。

【0337】本発明の第7実施形態

動的変換のためのページフォールト回復

第7実施形態の概容

動的変換は、1つのマシン語で書いたコンピュータプログラムを、そのプログラムの実行中に、他のマシン語で書いたものに変換する動作である。動的変換プログラムは変換元マシン命令をターゲットマシン命令に変換する前に、それらを読まなければならない。変換元マシン命令を読んでいる間、変換プログラムはページアウトしたメモリから読むことによって、ページフォールトを起こす可能性があるが、その場合、メモリ（実記憶装置）にページを移す（page in：ページインする）ことは効率的ではない。ここに述べる変換プログラムは、ページアウトデータを読むことなしに、ページフォールトから回復し、変換処理を継続する。

【0338】第7実施形態に関する図の説明

図32は本発明の第7実施形態に従って、変換元命令の通常実行時には起きないページフォールトを、如何にして動的変換プログラムがそれを起こす可能性があるかを示す。図33は変換処理中にページフォールトから回復し、変換処理を続行するための本発明の第7実施形態によるアルゴリズムを示す。

【0339】第7実施形態の詳細な説明

動的変換プログラムは、実際に実行される命令のサクセサだけではなく、命令の可能なサクセサの全てを読むから、物理メモリへのコピーに際して、悪い対象であるページにアクセスする可能性がある。例えば、図32に示すように、条件付きブランチ命令は2つのサクセサ、即ちフォールスルーサクセサとブランチテイクンサクセサ（branch taken successor）を有している。CPUが条件付きブランチ命令を実行するとき、もしブランチが取られてなければ、ブランチテイクンサクセサがロードされることは決してない。それ故、ページフォールトは起こらない。動的変換プログラムがブランチ命令を読む際、変換プログラムはどちらのサクセサが実際に実行されるかを知らずに、フォールスルー及びブランチテイクンサクセサの両方を読もうとする。たとえブランチサク

セサが決して実行されなくても、それを読むことがページフォールトを引き起こす可能性がある。

【0340】ページフォールトを扱う通常の方法は、要求されたメモリにページインし、ソフトウェアでメモリアクセスを行い、それから欠陥命令の後に実行を継続する。この方法は2つのコストを伴う。第1は、時間が掛かる。即ち、物理メモリから1つのページを補助記憶に移し、別のページを補助記憶から物理メモリに移し、それからメモリアクセスを行うのに時間が掛かる。第2に、ページインされるメモリのページセットを変えることになる。物理メモリにコピーされるページは、それが再度ページアウトされるまでは、頻繁にはアクセスされないかもしれない。このことは、それを物理メモリにコピーすることが悪い考えであったことを意味する。

【0341】動的変換プログラムは頻繁にページフォールトを起こす可能性があるから、それらページフォールトに掛かる費用を低減することは有益である。動的変換プログラムは、新たなページを物理メモリにコピーしないこと、そして既に物理メモリにあるページを取り去らないようにすることで、余分なページフォールトに掛かる費用を最小にする。こうすることによって、コピー時間を節約し、また、まれにしか参照されないページをコピーすることが確実になくなる。ページをコピーする代わりに、ページフォールトハンドラは、変換プログラムの指定するチェックポイントに戻す。

【0342】変換プログラムは基本ブロックと言う単位から変換元命令を読み取る。もし1つの基本ブロックを読んでいる間にページフォールトが起きたら、変換プログラムはそのブロックを無視し、他の何れかのブロックの変換を続行する。基本ブロックの全てを読んだ後、それらは一組のターゲット命令に変換される。ページフォールトを起こしたブロックを無視する方法を図33に示す。基本ブロックを読む前に、変換プログラムはチェックポイントを作る。チェックポイント以前の基本ブロックの読取りは全て安全で、チェックポイントの後で起こるページフォールトによる影響を何ら受けることはない。変換プログラムは次の基本ブロックを読み掛かる。もしそこでページフォールトが起きたら、直ちにチェックポイントにジャンプする。こうして、その基本ブロックをスキップして、次の基本ブロックを読むようにする。

【0343】第7実施形態の特別目的

本発明の第7実施形態は、メモリアクセスが不首尾の際、ページを物理メモリにコピーすることを止める一方で、変換を続けさせ、動的変換に於けるメモリアクセスコストを低減する方法である。

【0344】第7実施形態の要約

ここで述べたページフォールト回復機構は、非物理的にマッピングしたメモリにアクセスする際の、動的変換コ

ストを低減する方法である。この方法は、ページフォールトのために、変換元マシン命令の全てを読むことができない時でさえ、動的変換の継続を可能にする。この発明は、メモリアクセスが不首尾の際、ページを物理メモリにコピーすることを止める一方で、変換を続けさせ、動的変換に於けるメモリアクセスコストを低減する方法である。

【0345】本発明の第8実施形態

動的変換のための変換されたコードからの出口記録

第8実施形態の概容

動的変換は、1つのマシン語で書いたコンピュータプログラムを、そのプログラムの実行中に、他のマシン語で書いたものに変換する動作である。動的変換プログラムは命令の実行中にその命令をプロファイリングすることによって、命令を選択し、変換する。頻繁に実行された命令は変換されるが、たまにしか実行されないものは変換されない。変換された命令は、プロファイラによる幾つかの命令の見落とし、即ち、頻繁に解釈実行された命令の見落としを起こす可能性がある。変換されたコードからの特定の出口を記録することによって、頻繁に実行された命令の全てをプロファイルし、それら命令を全て確実に変換することができる。

【0346】第8実施形態に関する図の説明

図34は本発明の第8実施形態によるブランチプロファイラを有する動的変換システムに於ける制御フローのパターンを示す。

【0347】第8実施形態の詳細な説明

“動的変換のためのブランチロッガー”と題して記載したように、動的変換システムは、オリジナルプログラムのブランチ命令を解釈する際、それらをプロファイルし、どの命令が頻繁に実行され、どの命令が頻繁には実行されないかを決定する。ブランチロッガーはブランチ命令を単にプロファイルするだけで、全ての頻繁に実行される命令は、頻繁に実行されるブランチを介して到達可能であると仮定する。或る場合には、プロファイルされたブランチを実行せずに、制御フローが変換された命令から解釈された命令に戻されるから、動的変換プログラム自身、この仮定を真ではないとする。変換プログラムはこうした場合を識別することができ、この制御フローを、それがあたかもブランチであるかのようにプロファイルする特定の変換された命令を生成する。

【0348】図34は、制御がどの様にして解釈された命令から変換された命令に流れ、そしてまた、その逆に流れるかを示す。制御が変換された命令から出るときは何時でも、変換プログラムは出口があたかもブランチ命令であるかのようにプロファイルされていることを確かめる。制御が変換された命令から解釈された命令に流れる場合が幾つかある。

【0349】その第1は、非固定変換先へのブランチがある場合である。変換プログラムは、そのブランチの後

で、どの命令が実行されるかを知らないで、その命令と同じ変換ユニットにブランチとして組み合わせることができない。その代わりに、変換プログラムは、変換されたコードから解釈されたコードに戻る出口を生成する。

【0350】第2は、変換中のページフォールトのために、読むことができない命令がある場合である。“動的変換のためのページフォールト回復”と題して記載したように、変換プログラムは、ページフォールトのために読むことのできない命令ブロックを無視する。変換されたプログラムは、それらのブロックに到達したとき、ジャンプして解釈された命令に戻らなければならない。

【0351】第3は、変換が行われているときに、たまにしか実行されない命令が幾つかある場合である。“動的変換のためのブロックピッキング閾値”と題して記載したように、そうした命令はたまにしか実行されないから、変換されない。しかし、それらの命令は将来、頻繁に実行されるかもしれない。そこで、変換プログラムはそれら命令に対する出口を記録しなければならない。この特徴によって、動的変換システムは、頻繁に実行される命令の分布を変える実行パターンの変更に適応することができるようになる。

【0352】変換されたコードからの出口が記録されるので、より多くの命令が変換される。このことは、命令の変換されたバージョンが存在する機会を増加させる。それ故、動的変換システムを長時間実行した後、1つの変換されたユニットからの出口の殆どが、解釈されたコードへ戻すジャンプの代わりに、他の変換されたユニットへジャンプを起こす。このことは、高速変換された命令をより頻繁に使用することからくる直接利益と共に、ブランチロギング命令を頻繁には実行しないことによる間接利益も持っている。

【0353】第8実施形態の特別目的

本発明の第8実施形態は、変換されたユニットの可能な出口をプロファイリングすることによって、たとえ頻繁に実行される命令がプロファイルされた何れのブランチを通しても到達されない場合でも、それを確実に変換する方法を目指している。

【0354】第8実施形態の要約

動的変換システムは、頻繁に実行される命令の全てを探して変換しなければならない。これは、ブランチ命令をプロファイリングすることによって完成される。しかし、命令の変換は、プロファイルされたブランチを含まない命令への経路を生成する。それ故、プロファイリングは変換された命令からの出口を含むように広げられる。この発明は、変換されたユニットの可能な出口をプロファイリングすることによって、たとえ頻繁に実行される命令がプロファイルされた何れのブランチを通しても到達されない場合においても、それを確実に変換する方法である。

【0355】本発明の第9実施形態

動的変換のためのブロックピッキング閾値

第9実施形態の概容

動的変換は、1つのマシン語で書いたコンピュータプログラムを、そのプログラムの実行中に、他のマシン語で書いたものに変換する動作である。動的変換プログラムは、頻繁に実行される変換元プログラム部分を全て変換し、たまにしか実行されない部分の全てを無視しなければならない。このことを達成するため、変換システムはブランチ命令をプロファイルし、実行確率が特定の閾値以下の命令に関しては変換を行わない。

【0356】第9実施形態に関する図の説明

図35は、本発明の第9実施形態に従って、動的変換プログラムが如何にブランチプロファイル情報を用いて、基本ブロックの実行確率を計算するかを示す。

【0357】第9実施形態の詳細な説明

動的変換プログラムの目的は、1つのコンピュータプログラムのオリジナル変換元言語命令を、さらに効率の良いターゲット言語命令に変換することによって、そのコンピュータ全体の実行速度を改善することである。動的変換の利点はオリジナルプログラムを実行するための合計時間を、そのプログラムの変換に要する時間と変換されたプログラムの実行時間の和と比較することによって計られる。プログラムのどの部分を変換するにしても、その変換に要する時間はほぼ一定であるから、1部を変換することの利益は、基本的にその部分を使用する回数によって決まる。頻繁に実行される命令は変換する価値があるが、まれにしか実行されない命令はその価値はない。

【0358】異なる命令の実行頻度を計るために、動的変換システムはブランチ命令をプロファイルすることができる。このプロファイル情報を用いれば、頻繁に実行される命令を選択することができ、その点で変換することができる。初期命令の後、変換プログラムは、たまにしか実行されないサクセサ命令を読まないようにして、できるだけ多くの頻繁に実行されるサクセサ命令を読もうとする。ブロックピッキング閾値は、サクセサ命令が頻繁に実行されているか、又はたまにしか実行されていないかを決定するのに使用される。

【0359】動的変換プログラムは、基本ブロックと称する単位で命令を読む。1つの基本ブロックでは、全ての命令は同じ回数だけ実行される。従って、命令は、その全てが頻繁に実行されるか、又はその全てがたまにしか実行されないかの何れかとなる。

【0360】動的変換プログラムはブランチ命令からの情報を用いて、基本ブロックが頻繁に実行されるか、たまにしか実行されないかを決定する。この処理を図35に示す。変換プログラムは、実行経路が第1の変換された命令から、与えられた基本ブロックに向けて取られる確率を計算する。第1の基本ブロックは、第1の命令を

含むから、それには100%の確率が与えられる。もしこの現在のブロックがサクセサだけしか持っていなければ、そのサクセサは現在のブロックと同じ実行確率を持っている。もし現在のブロックが条件付きブランチに終わっていれば、現在のブロックの確率は、ブランチプロファイル情報に従って2つのサクセサに分けられる。例えば、現在のブロックの実行確率が50%であって、ブランチ命令で40回実行され、10回取られて終われば、ブランチテイクンサクセサの確率は $50\% \times 25\% = 12.5$ 、フォールスルーサクセサの確率は $50\% \times 75\% = 37.5$ となる。

【0361】ブロックピッキング閾値と呼ばれる可変閾値は、頻繁に実行されるブロックを選択するのに使用される。もしブロックの実行確率がこの閾値より大きい、又は等しければ、そのブロックは高い頻度で使用されると考えて、変換される。またもしブロックの実行確率がこの閾値未満であれば、そのブロックは低い頻度でしか使用されないと考えて、変換さない。

【0362】このブロックピッキング方法の1つの重要な特性は、選択されたブロックセットが接続されることである。この実行確率を計算する方法として、更に複雑な方法、例えば全ての手続きからの確率を加算する等の方法がある。しかし、この方法では、ブロックセットは不連続なものとなる。不連続なブロックセットを変換することは可能だが、もしそれらが全て接続されていれば、変換されたコードを最適化するもっと多くの機会が得られる。

【0363】第9実施形態の特別目的

本発明の第9実施形態は、変換に当たって、たまにしか実行されないブロックから頻繁に実行されるブロックを分離する実行確率の閾値を用いて、頻繁に実行される命令のブロックを選択すると共に、たまにしか実行されない命令のブロックを無視することによって、動的変換効率を改善する方法を目指す。

【0364】第9実施形態の要約

動的変換システムのコストは、変換された命令の数に比例すると共に、その利益は変換された命令が実行される回数に比例する。それ故、頻繁に実行される命令だけを変換し、たまにしか実行されない命令を無視することが最も効率的である。本発明は、変換に当たって、たまにしか実行されないブロックから頻繁に実行されるブロックを分離する実行確率の閾値を用いて、頻繁に実行される命令のブロックを選択すると共に、たまにしか実行されない命令のブロックを無視することによって、動的変換効率を改善する方法である。

【0365】これまで、本発明に関する幾つかの好適実施形態について、それらを図示し、説明してきたが、これら実施形態について、本発明の原理及び精神から逸脱することなく変更が可能であることは、当業者の理解するところであろう。本発明の範囲は、請求の範囲の記載

及びそれと均等なものにある。

【0366】以上の説明により本発明は次のような特徴を有する。

(1) 変換先コンピュータアーキテクチャシステム上で変換元コンピュータアーキテクチャをエミュレートするコンピュータアーキテクチャエミュレーションシステムであって、変換元オブジェクトコードを対応する変換されたオブジェクトコードにそれぞれ変換し、該変換元オブジェクトコードのブランチ命令の実行数を決定するインタプリタと、対応するブランチ命令の実行数が閾値を越えたときに該変換元オブジェクトコードの命令をセグメントにグループ化し、該セグメントを動的にコンパイルするコンパイラと、を具備するコンピュータアーキテクチャエミュレーションシステム。

【0367】(2) コンパイルされないセグメントに対応するブランチ命令は、メモリに記憶される、前記

(1) に記載のコンピュータアーキテクチャエミュレーションシステム。

【0368】(3) 該閾値を越えなかったブランチ命令に対応するセグメントは、コンパイルされない、前記

(2) に記載のコンピュータアーキテクチャエミュレーションシステム。

【0369】(4) 前記インタプリタが、該変換されたオブジェクトコード命令を実行している間に、コンパイルされないブランチ命令に対応するセグメントは、メモリに記憶される、前記(1)に記載のコンピュータアーキテクチャエミュレーションシステム。

【0370】(5) 前記インタプリタ及び前記コンパイラは、実時間でマルチタスキングオペレーティングシステムにて同時に動作するタスクである、前記(1)に記載のコンピュータアーキテクチャエミュレーションシステム。

【0371】(6) 前記インタプリタによって決定されたブランチ命令のブランチプロファイル情報を記憶するブランチローガーを更に具備する、前記(1)に記載のコンピュータアーキテクチャエミュレーションシステム。

【0372】(7) 前記ブランチプロファイル情報は、ブランチアドレス、ブランチサクセサ、非ブランチサクセサ、ブランチ実行カウント、及びブランチテイクンカウントを含み、前記ブランチプロファイル情報は、ブランチ命令エミュレーションの間に前記インタプリタによって記録される、前記(6)に記載のコンピュータアーキテクチャエミュレーションシステム。

【0373】(8) 変換可能な命令へのジャンプ又はそれからのジャンプを実行するブランチ命令の後にコードフラグが置かれ、該対応するコードフラグを参照することにより、該対応するブランチ命令に対するサクセサ命令が変換可能か否かを決定すべくチェックされる、前記(1)に記載のコンピュータアーキテクチャエミュレーションシステム。

【0374】(9) ブランチ命令に対するサクセサ命令の実行数が対応する閾値を上回ったとき、該ブランチ命令の最初の変換が実行される、前記(1)に記載のコンピュータアーキテクチャエミュレーションシステム。

【0375】(10) 頻繁にブランチされる命令に対応するセグメントの変換を開始するため、前記インタプリタが変換元オブジェクトコードのエミュレーションを継続している間、前記インタプリタと前記コンパイラとが通信する、前記(1)に記載のコンピュータアーキテクチャエミュレーションシステム。

【0376】(11) 変換されるべきセグメントを記憶する待ち行列が所定の容量に達したとき、閾値を上げることによって、コンパイルされるべきセグメントのコンパイル率が制御される、前記(1)に記載のコンピュータアーキテクチャエミュレーションシステム。

【0377】(12) 前記コンパイラは、コンパイルが開始されたアドレスに対応するプロファイルを使用して、メモリに記憶されている各命令を順次追跡する間に、最適化されたオブジェクトコードを生成する、前記(1)に記載のコンピュータアーキテクチャエミュレーションシステム。

【0378】(13) 前記コンパイラは、ブロックが起こしたページフォールトを検出した際にはブロックをコンパイルせず、ブランチロガーにブランチ情報を記録するためのオブジェクトコードを作成する、前記(12)に記載のコンピュータアーキテクチャエミュレーションシステム。

【0379】(14) 命令実行処理が所定の実行率でタイムリーに実行されていない場合、前記コンパイラは、プロファイルを用いて実行状態を追跡し、ブランチカウンタが所定数を下回っているか否かをチェックし、ブランチ情報を記録するためオブジェクトコードを作成する、前記(13)に記載のコンピュータアーキテクチャエミュレーションシステム。

【0380】(15) 実行数を含む変換元オブジェクトコードにおけるブランチ命令のプロファイル情報を記憶するとともに、頻繁に実行されるブランチ命令のプロファイル情報を記録するキャッシュと、頻繁には実行されないブランチ命令のプロファイル命令を記憶するブランチログと、を含むブランチロガー、を更に具備する、前記(1)に記載のコンピュータアーキテクチャエミュレーションシステム。

【0381】(16) プロファイル情報は、ブランチアドレス情報とブランチ変換先情報とを組み合わせてキャッシュに組織される、前記(15)に記載のコンピュータアーキテクチャエミュレーションシステム。

【0382】(17) キャッシュに組織されるプロファイル情報は複数のグループに記憶され、各グループはプロファイル情報のエントリの降順にそれぞれのグループ内に組織される、前記(16)に記載のコンピュータア

ーキテクチャエミュレーションシステム。

【0383】(18) 各ブランチ命令はシードであり、前記コンパイラは、変換元オブジェクトコードのセグメントを選択して、該シード及び該ブランチのプロファイル情報に基づいてコンパイルするブロックピッカと、該セグメントを命令の線形リストへと平坦化するブロックレイアウトユニットと、オリジナル命令を変換されたコードセグメント命令に実際にコンパイルする最適化コード発生ユニットと、を更に含む、前記(1)に記載のコンピュータアーキテクチャエミュレーションシステム。

【0384】(19) ブロックピッカは、オリジナル命令を記述する制御フローグラフを生成してコンパイルし、該制御フローグラフをブロックレイアウトユニットに渡す、前記(18)に記載のコンピュータアーキテクチャエミュレーションシステム。

【0385】(20) 変換先コンピュータアーキテクチャシステム上で変換元コンピュータアーキテクチャをエミュレートするコンピュータアーキテクチャエミュレーションシステムであって、変換元オブジェクトコードを、対応する変換されたオブジェクトコードに、それぞれ変換すると共に、それぞれが変換されたオブジェクトコード命令の実行の間に、実時間で変換元オブジェクトコードのブランチ情報をプロファイルする複数のインタプリタと、前記複数のインタプリタの何れかからの変換元オブジェクトコード命令を、変換元オブジェクトコードに於ける対応するブランチ命令に基づいてセグメントにグループ化し、対応するブランチ命令の実行数が閾値より大きいとき、変換元オブジェクトコードのセグメントを動的にコンパイルするコンパイラと、を具備するコンピュータアーキテクチャエミュレーションシステム。

【0386】(21) 前記複数のインタプリタの各々は、ブランチ命令をプロファイルすると共に、閾値を越えなかったブランチ命令を、ブランチロガーをコールして記憶する、前記(20)に記載のコンピュータアーキテクチャエミュレーションシステム。

【0387】(22) 変換先コンピュータアーキテクチャシステム上で変換元コンピュータアーキテクチャをエミュレートするコンピュータアーキテクチャエミュレーションシステムであって、変換元オブジェクトコードを対応する変換されたオブジェクトコードにそれぞれ変換するインタプリタであって、変換元オブジェクトコードのブランチ命令を、各ブランチ命令に関する実行数を記憶すると共にその実行数を閾値と比較することによって、プロファイルし、閾値を越えたブランチ命令をシードとして指定するインタプリタと、該シードに基づいて、変換元オブジェクトコード命令をセグメントにグループ化し、前記インタプリタによる変換及びプロファイルリングの間に、変換元オブジェクトコードのセグメントを動的にコンパイルするコンパイラと、を具備するコンピュータアーキテクチャエミュレーションシステム。

【0388】(23) 各セグメントは、対応するシードに基づいて変換元オブジェクトコードを最適化した結果得られた命令を含み、各セグメントは、単位として導入及び非導入される、前記(22)に記載のコンピュータアーキテクチャエミュレーションシステム。

【0389】(24) コンパイルされないセグメントに対応するブランチ命令はメモリに記憶され、閾値を越えないブランチ命令に対応するセグメントはコンパイルされない、前記(23)に記載のコンピュータアーキテクチャエミュレーションシステム。

【0390】(25) 前記インタプリタによって決定されたブランチ命令のブランチプロファイル情報を記憶するブランチローガーを更に具備し、該ブランチプロファイル情報は、ブランチアドレス、ブランチサクセサ、非ブランチサクセサ、ブランチ実行カウント、及びブランチテイクンカウントを含むとともに、ブランチ命令のエミュレーションの間に、前記インタプリタによって記録される、前記(23)に記載のコンピュータアーキテクチャエミュレーションシステム。

【0391】(26) 変換可能な命令へのジャンプ又はそれからのジャンプを実行するブランチ命令の後にコードフラグが置かれ、対応するコードフラグを参照して、対応するブランチ命令が変換可能か否かを決定するために、サクセサ命令がチェックされる、前記(23)に記載のコンピュータアーキテクチャエミュレーションシステム。

【0392】(27) ブランチ命令に対するサクセサ命令の実行数が閾値を上回ったときにブランチ命令が最初に変換される、前記(23)に記載のコンピュータアーキテクチャエミュレーションシステム。

【0393】(28) 変換されるべきセグメントを記憶する待ち行列が所定の容量に達したとき、閾値を上げることによって、セグメントのコンパイル率が、コンパイルされるべく制御される、前記(23)に記載のコンピュータアーキテクチャエミュレーションシステム。

【0394】(29) 命令実行処理が所定の実行率でタイムリーに実行されていない場合に、前記コンパイラは、プロファイルを用いて実行状態を追跡し、ブランチカウントが所定数を下回っているか否かをチェックし、ページフォールトのようなブランチ情報を記録するためのオブジェクトコードを作成する、前記(23)に記載のコンピュータアーキテクチャエミュレーションシステム。

【0395】(30) 実行数を含む変換元オブジェクトコードにおけるブランチ命令のプロファイル情報を記憶するブランチローガーであって、頻繁に実行されるブランチ命令のプロファイル情報を記憶するキャッシュと、頻繁には実行されないブランチ命令のプロファイル命令を記憶するブランチログと、を含むブランチローガー、を更に具備し、プロファイル情報は、ブランチアドレス情報と

ブランチ変換先情報とを組み合わせるキャッシュに組織されるとともに、該プロファイル情報は、複数のグループに、該グループへのエントリの降順に記憶される、前記(23)に記載のコンピュータアーキテクチャエミュレーションシステム。

【0396】(31) 前記コンパイラは、変換元オブジェクトコードのセグメントを選択し、該シード及び該ブランチのプロファイル情報に基づいてコンパイルするブロックピッカであって、オリジナル命令を記述する制御フローグラフを生成し、そのグラフをコンパイルするブロックピッカと、該制御フローグラフを命令の線形リストへと平坦化するブロックレイアウトユニットと、オリジナル命令を変換されたコードセグメント命令に実際にコンパイルする最適化コード発生ユニットと、を更に含む、前記(23)に記載のコンピュータアーキテクチャエミュレーションシステム。

【0397】(32) 多重タスキング変換先コンピュータアーキテクチャ上で変換元コンピュータアーキテクチャをエミュレートする多重タスキングコンピュータアーキテクチャエミュレーションシステムであって、変換元オブジェクトコードを対応する変換されたオブジェクトコードにそれぞれ変換し、変換元オブジェクトコードのブランチ命令の実行数を決定するインタプリタタスクと、多重タスキング変換先コンピュータアーキテクチャ上で前記インタプリタタスクと共に動作するコンパイラタスクであって、対応するブランチ命令の実行数が閾値を越えたとき変換元オブジェクトコードの命令をセグメントにグループ化し、このセグメントを動的にコンパイルするコンパイラタスクと、を具備する多重タスキングコンピュータアーキテクチャエミュレーションシステム。

【0398】(33) 前記多重タスキングコンピュータアーキテクチャエミュレーションシステムは動的変換システムであり、前記多重タスキングコンピュータアーキテクチャシステムは、前記インタプリタタスクによって送られるコンパイル要求の率と、前記コンパイラタスクによって完成されるコンパイルの率とを、閾値を変えることによってコンパイラタスクが遊休状態に入らないようにしつつ、等しくするソフトウェアフィードバック、を更に具備する、前記(32)に記載の多重タスキングコンピュータアーキテクチャエミュレーションシステム。

【0399】(34) 前記コンパイラタスクによってコンパイルされるべきセグメントを記憶する待ち行列を更に具備し、閾値は前記コンパイラタスクをon又はoffする最小閾値と比較される、前記(33)に記載の多重タスキングコンピュータアーキテクチャエミュレーションシステム。

【図面の簡単な説明】

【図1】本発明の好適実施形態によるOOCシステムの高

レベルアーキテクチャを示すブロック図である。

【図2】最適化オブジェクトコード変換の主要構成要素を、オリジナルコードの1つのセクションをコンパイルするための制御フローと共に示す流れ図である。

【図3】通常の実行時に於ける最適化オブジェクトコード変換の制御フローを示す流れ図である。

【図4】各種変数を設定した時のOOCTバッファの概略図である。

【図5】(a)、(b)及び(c)は変換テーブルの構造を示す概略図である。

【図6】インタプリタがセグメントに入り、そして其処から出る過程を示すブロック図である。

【図7】セグメントを生成し、インタプリタによるセグメントへの到達を可能とし、旧セグメントには到達不能とし、そして旧セグメントを削除するコンパイル方法を示すブロック図である。

【図8】BRANCH_RECORDの構造を示すブロック図である。

【図9】BRANCH_RECORDs を記憶する大きいハッシュテーブルの一部としてのブランチログの構造を示す概略図である。

【図10】BRANCH_L1_RECORDs の2次元配列であるL1キャッシュを示す概略図である。

【図11】インタプリタによるL1キャッシュの動作を実行する方法を示す概略図である。

【図12】本発明の実施形態によるコンパイラの全体構造を示す概略図である。

【図13】本発明の実施形態によるブロックピッカの例を示す概略図である。

【図14】ENTRY 命令とGOTO命令の間にフィルを挿入した2つの外部入力点を備えたコードアウトラインを示すブロック図である。

【図15】OASSIGN 挿入例を示すブロック図である。

【図16】デッドコード削除及びアドレスチェック削除の例を示すブロック図である。

【図17】アドレスチェック削除の例を示すブロック図である。

【図18】共通部分式削除(“CSE”)の例を示すブロック図である。

【図19】コピー伝播の例を示すブロック図である。

【図20】定数の畳込み例を特に示す。

【図21】本発明の実施形態による比較インフラストラクチャを有する上記処理の例を特に示す。

【図22】異なる周囲命令に対して同じ命令のためのコードを発生するコード発生例を特に示す。

【図23】本発明の第2実施形態による動的最適化オブジェクトコード変換に用いるシステム構成を示す。

【図24】本発明の第3実施形態による同時動的変換に

用いるシステム構成を示す。

【図25】本発明の第3実施形態に従ってインタプリタとコンパイラを、例えば1つのタスク実行に際して結合する場合と、例えばそれらを異なるタスク実行に際して分離する場合との差を示す。

【図26】本発明の第4実施形態に従って、どの命令が変換可能であり、どの命令が変換不能であるかを記録するのに使用する変換テーブルを示す。

【図27】本発明の第4実施形態による方法が、エミュレータに関するプロファイリング負担を如何に低減するかを示す。

【図28】本発明の第5実施形態によりインタプリタとコンパイラを分離した動的変換システムの全体構造を示す。

【図29】本発明の第5実施形態によるソフトウェアアードバック機構の構成要素を示す。

【図30】本発明の第6実施形態によって、変換タスク実行中に、如何に待ち行列を用いて変換要求を保持するかを示す。

【図31】本発明の第6実施形態によって、如何にOOCTが要求する待ち行列が、安価な共用メモリ要求と、システムコール要求とを組み合わせるかを示す。

【図32】本発明の第7実施形態によって、変換元命令の通常実行時には起きないページフォールトを、如何にして動的変換プログラムがそれを起こす可能性があるかを示す。

【図33】変換処理中にページフォールトから回復し、変換処理を続行するための本発明の第7実施形態によるアルゴリズムを示す。

【図34】本発明の第8実施形態によるブランチプロファイラを有する動的変換システムに於ける制御フローのパターンを示す。

【図35】本発明の第9実施形態によって、動的変換プログラムが如何にブランチプロファイル情報を用いて、基本ブロックの実行確率を計算するかを示す。

【符号の説明】

100…OOCTシステム

104…コンパイラ

108…コンパイルされたコードセグメント

110…インタプリタ

112…ブランチロッガー

114…ブロックピッカ

116…ブロックレイアウトユニット

118…最適化コード発生ユニット

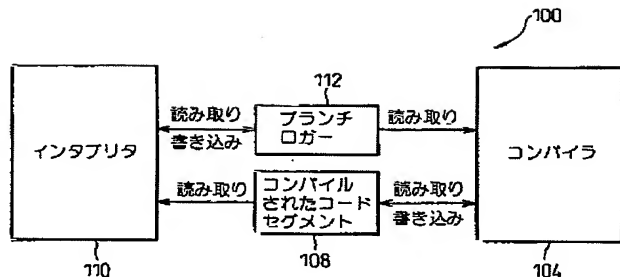
120…セグメント導入ユニット

124…IL(中間言語)発生器

126…オブティマイザ

128…コード発生器

【図1】



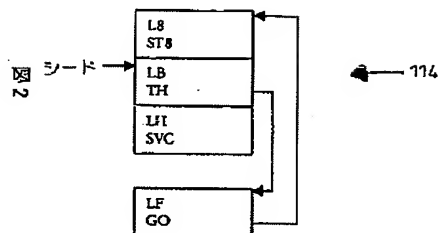
【図8】

図 8

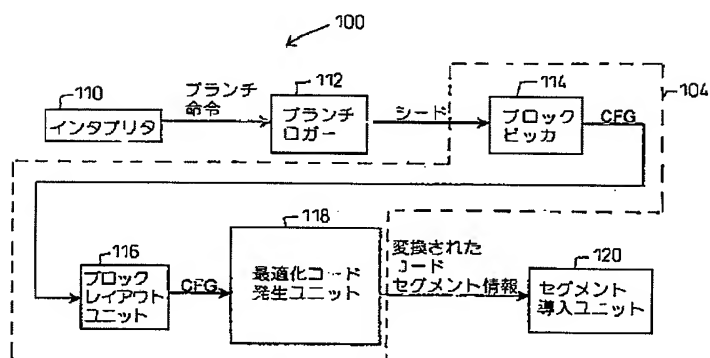
branch_address
branch_destination
branch_fall_through
encountered_count
taken_count
next

【図13】

図13



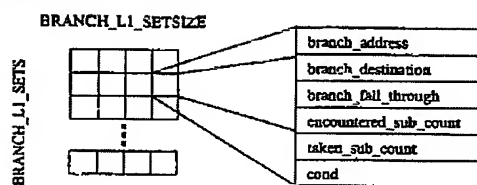
【図2】



【図3】

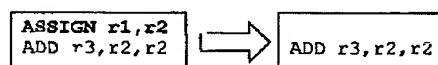
【図10】

図 10



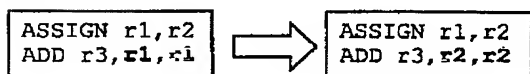
【図16】

図15

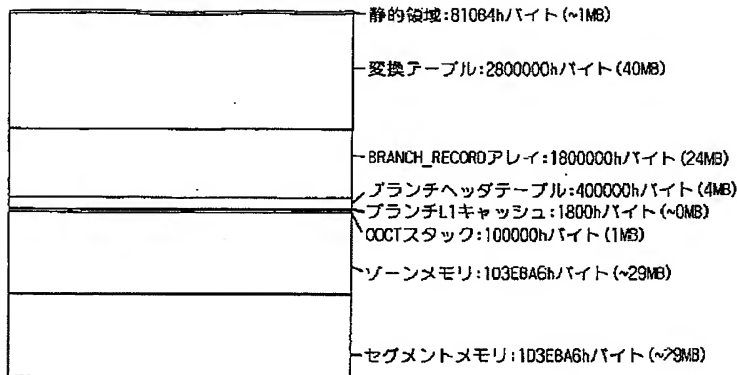


【図19】

図19



【図4】



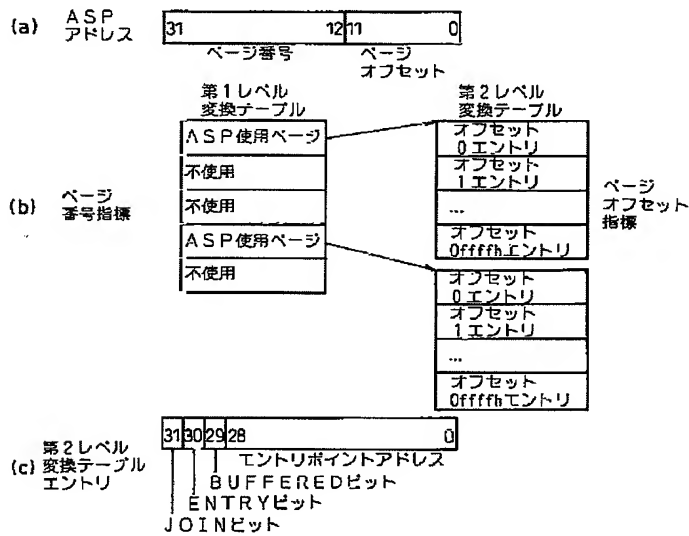
【図14】

```

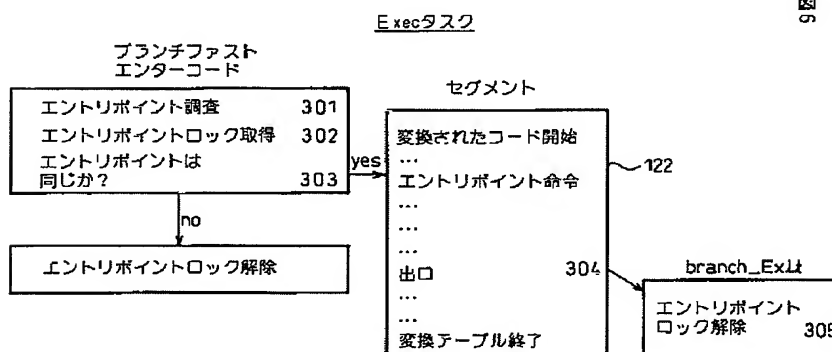
LABEL L1
...
LABEL L2
...
IGOTO
ENTRY 1
ASSIGN r1 r35
ASSIGN r2 r36
GOTO L1
ENTRY 2
ASSIGN r1 r35
GOTO L2

```

【図5】

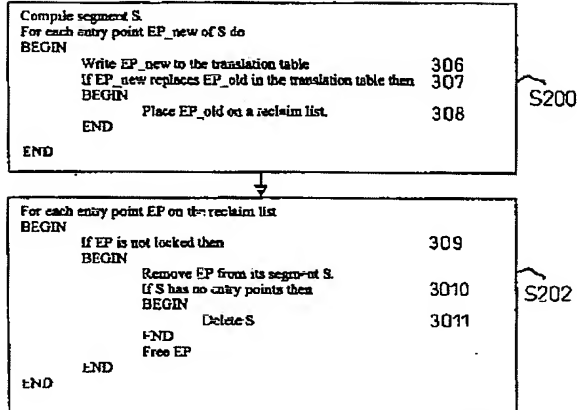


【図6】



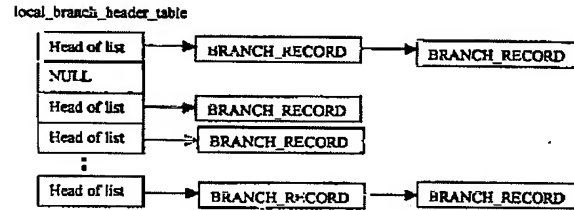
【図7】

図7 コンパイラタスク

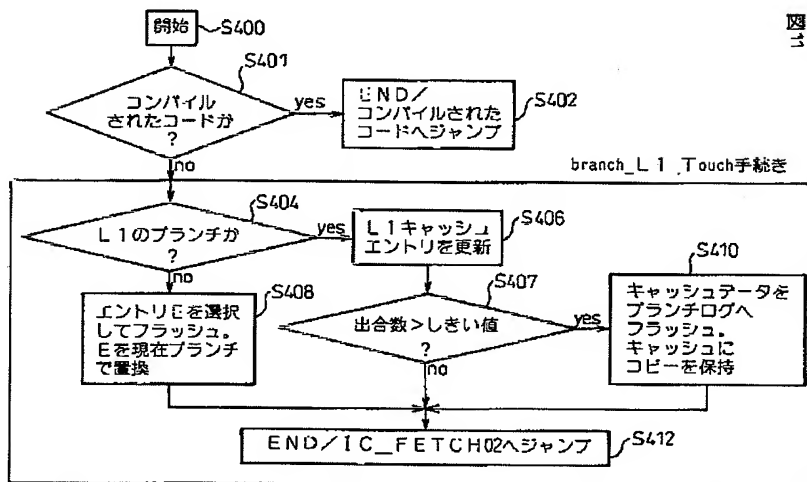


【図9】

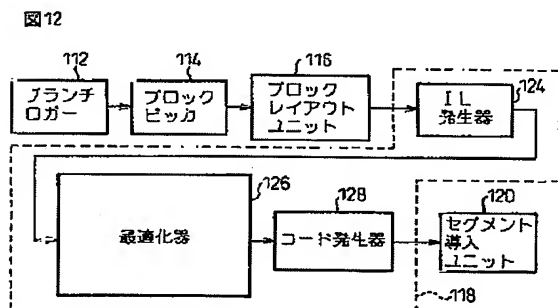
図9



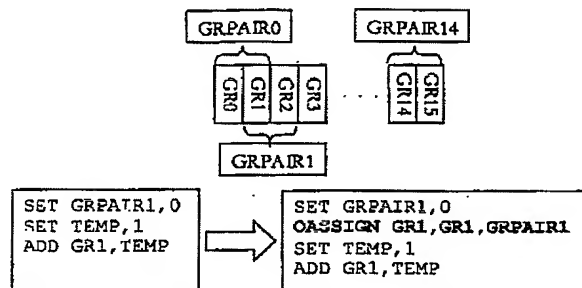
【図11】



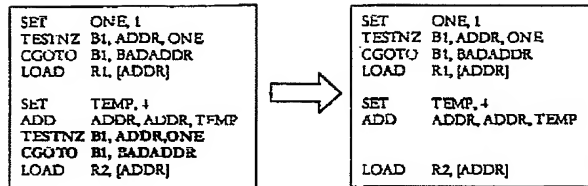
【図12】



【図15】



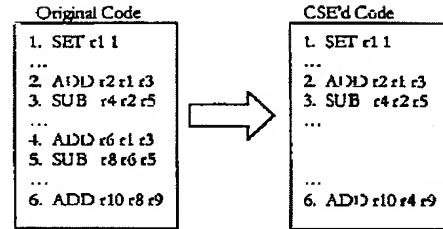
【図17】



【図18】

図17

図18



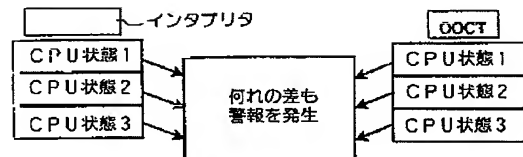
【図20】

【図21】

図20

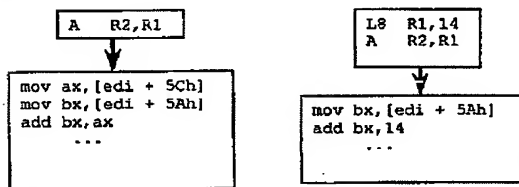


図21



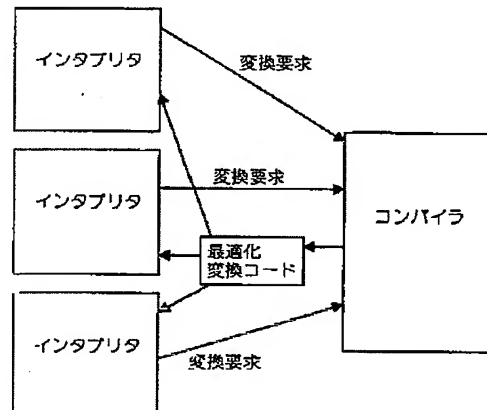
【図22】

図22



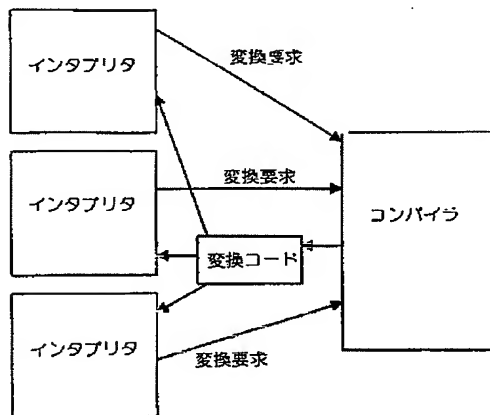
【図23】

図23



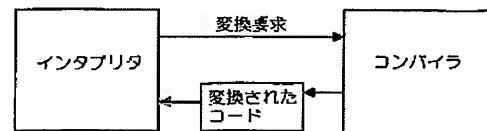
【図24】

図24

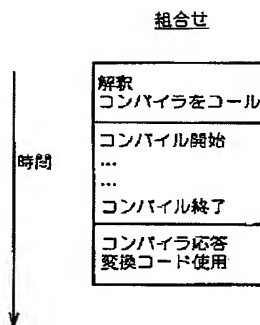


【図28】

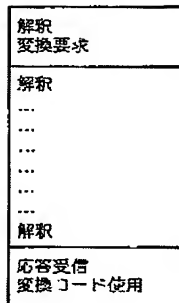
図28



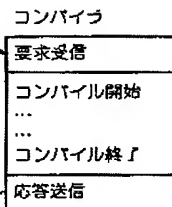
【图25】



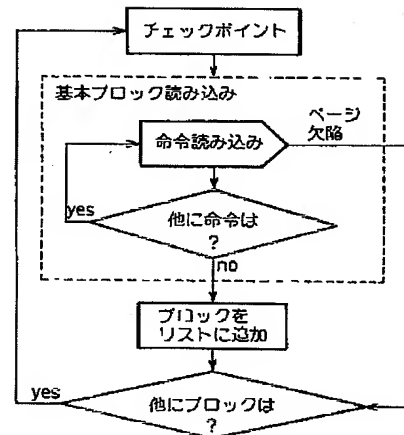
インタプリタ



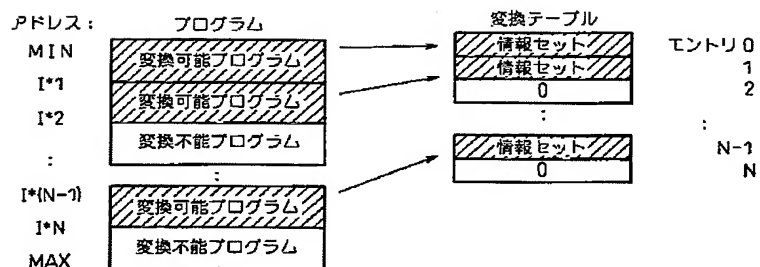
分離



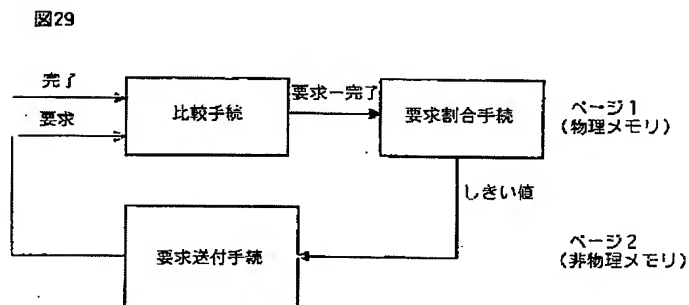
【図33】



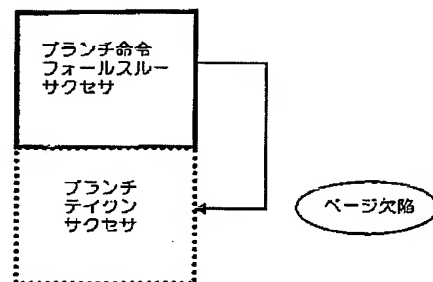
【図26】



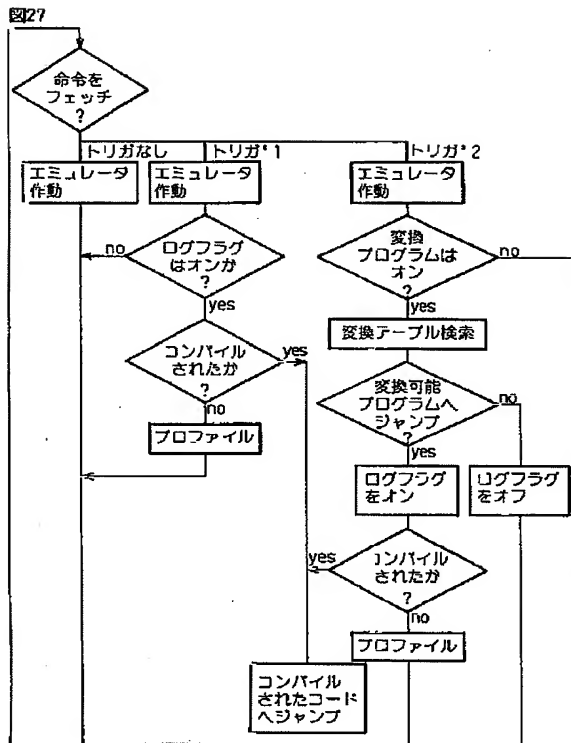
【图29】



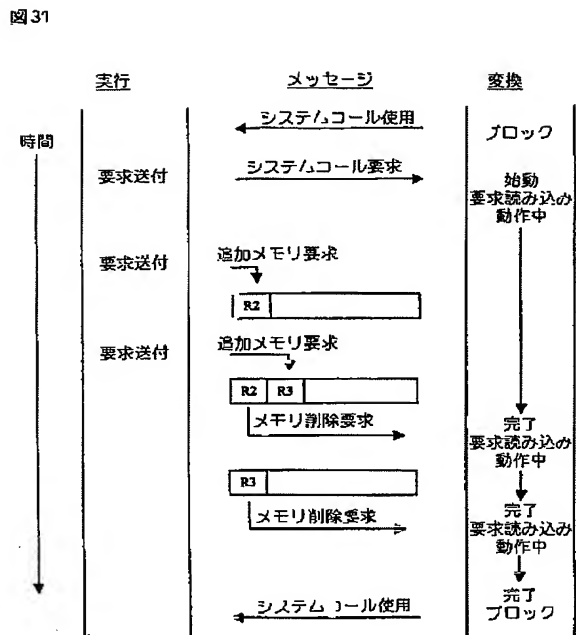
【図32】



【図27】



【図31】



【図30】

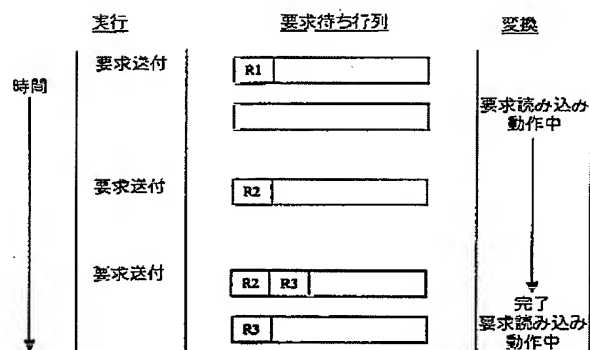


図30

【図34】

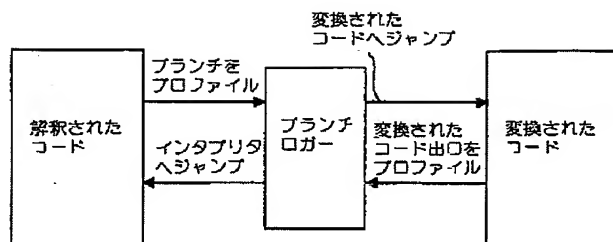


図34

【図35】

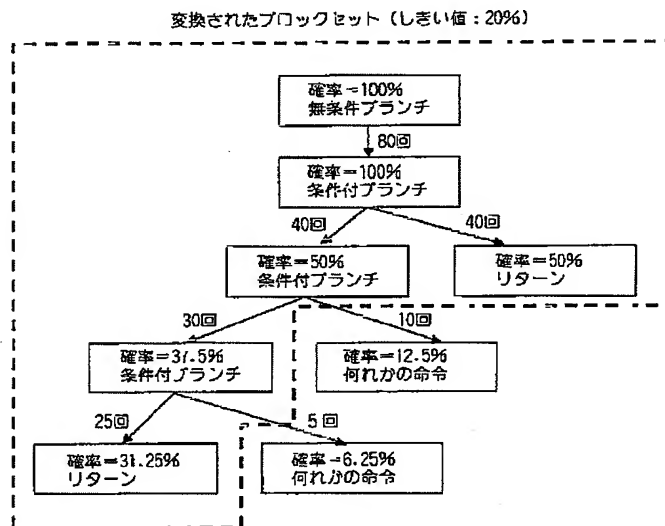


図35

フロントページの続き

(72)発明者 ジョセフ エー. バンク
アメリカ合衆国, ニューヨーク 10016,
ニューヨーク, イースト トゥエンティエ
イス ストリート 114, ナンバー2シー

(72)発明者 チャールズ ディー. ガレット
アメリカ合衆国, ワシントン 98115, シ
アトル, ブルックリン アベニュー ノース
イースト 6320

(72)発明者 和田 美加代
神奈川県川崎市中原区上小田中4丁目1番
1号 富士通株式会社内

(72)発明者 櫻井 三男
神奈川県川崎市中原区上小田中4丁目1番
1号 富士通株式会社内